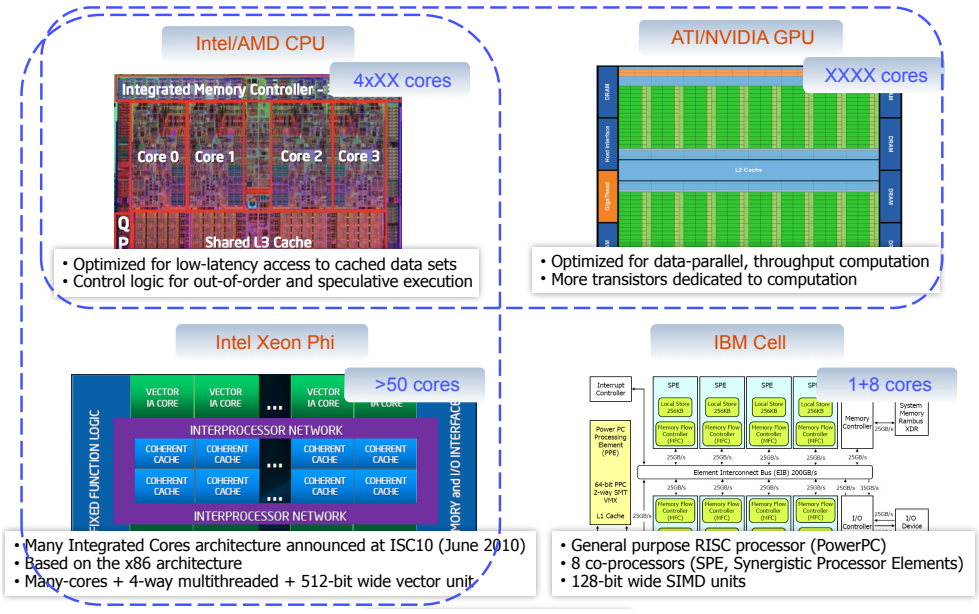
Goethe University of Frankfurt am Main

High Performance Computing

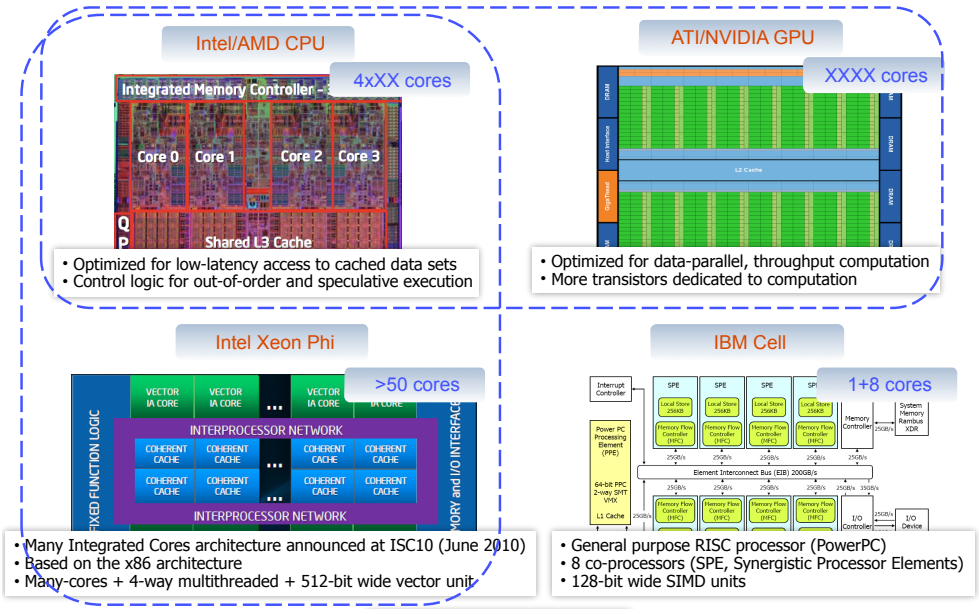
Practical Course



Goethe University of Frankfurt am Main

High Performance Computing

Practical Course



Prof. Dr. I. Kisel and PhD students V. Akishina, I. Kulakov, M. Zyzak

2012 - 2014

This course has been given at the Goethe University of Frankfurt am Main starting from summer semester 2012. The idea of the course is to give students practical experience and insight into parallel programming and many-core CPU/GPU/Phi computer architectures.

Table of contents

HPC Practical Course
Part 1

Introduction into Unix Shell and C++

V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak,

Goethe University of Frankfurt am Main

16 Feb 2014

Basic Unix Shell Commands

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems (Linux, OSX). There are different shell categories, for example Bourne shell (sh), Bourne-Again shell (bash), C shell (csh), TENEX C shell (tcsh). We consider bash commands, bash is often default shell, or can be called from other shells with 'bash' command.

pwd — current directory
cd [dirname] — change directory. '**cd ..**' - level up
mkdir [dirname] — create a new directory
ls [dirname] — list of files in the directory.
mv [fileOrDirName] [dirname] — moves a file or a directory to another directory.
mv [fileOrDirNameOld] [fileOrDirNameNew] — renames a file or a directory.
cp [filenameOld] [filenameNew] — copies a file.
cp -r [dirnameOld] [dirnameNew] — copies a directory and its content.
rm [filename] — removes a file.
rm -r [dirname] — removes a directory and its content.
grep [string] [filenames] — looks for the string in the files.
wget [fileurl] [dirname] — downloads file from web to the directory.
ssh [user]@[hostname] — connect into a remote machine.
scp [-r] [userOld]@[hostnameOld]:[fileOrDirNameOld] [userNew]@[hostnameNew]:[fileOrDirNameNew] — copies a file

or a directory and its content.
g++ [filenames].cpp [flags] -o [executablename] — compile C++ source code with GNU compiler and create executable. **./[executablename]** — run executable file.

The shell commands can be collected into script text files. Shell scripts have extension '**.sh**'.
Use '**. [bashScriptName].sh**' to execute bash script.

Types in C++

Type	Values	Size (bytes)
bool	“true” and “false”	1
unsigned char	0 … 255	1
signed char	–128 … 127	1
unsigned short int	0 … 65 535	2
short	–32 768 … 32 767	2
unsigned int	0 … 4 294 967 295	4
int	–2 147 483 648 … 2 147 483 647	4
float	3.4e–38 … 3.4e+38	4
double	1.7e–308 … 1.7e+308	8
long double	3.4e–4932 … 3.4e+4932	10 (12,16)

Type Casting

Implicit conversion **short a=200; float b; b = a;**	Does not require any operator, automatically performed when a value is copied to a compatible type. Exists for standard types. For users classes a constructor should be written.
Explicit conversion **b = (type) a;** **b = type (a);** **b = static_cast <new_type> (a)**	Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to, therefore is not safe, can lead to code that while being syntactically correct can cause runtime errors.
dynamic_cast <new_type> ()	Can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
static_cast <new_type> ()	Can perform conversions between pointers to related classes. Ensures that at least the classes are compatible if the proper object is converted. The overhead of the type-safety checks of dynamic_cast is avoided. A programmer should ensure the conversion is safe.
reinterpret_cast <new_type> ()	Converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
const_cast <new_type> ()	Manipulates the constness of an object, either to be set or to be removed.

More at: http://www.cplusplus.com/doc/tutorial%20%20/typecasting/

Conditional Operators

The conditional operator "if/else":

```
if(condition) {
 statement1;
}
else {
 statement2;
}
```

If condition is true, then statement1 is executed, if it is false – statement2, structure else is optional. Examples:

```
1) if (a>0) {
     b = a;
   } else
     b = 0;
   }
```

```
2) if (a>0)
     b = a;
```

The ternary conditional operator:

(condition) ? expression1 : expression2

If condition is true, then result is expression1, if it is false – expression2. Examples:

```
1) b = (a>0) ? a : 0;
```

```
2) b = (a>0) ? a : b;
```

More at: http://www.cplusplus.com/doc/tutorial/control/

The "for" Loop

for (initialisation; condition; increase) statement;

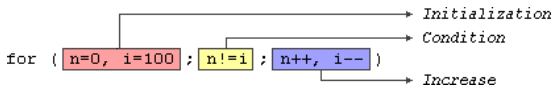
- Repeats statement while condition remains true.
- Provides specific locations to contain an initialisation statement and an increase statement.
- Is specially designed to perform a repetitive action with a counter which is initialised and increased on each iteration.

Examples:

```
1)for(int i=0; i<10; i++) {
  a[i] = i;
}
```

```
2) int i=0;
for(; i<10; i++) {
  a[i] = i;
}
```

```
3)          for ( n=0, i=100 ; n!=i ; n++, i-- ){
   // whatever here...
}
```



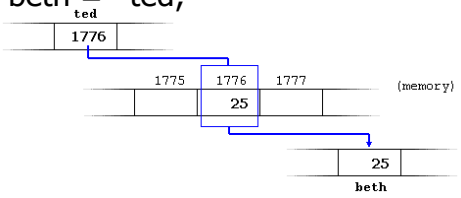
More at: http://www.cplusplus.com/doc/tutorial/control/

Pointers

The memory of a computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern.
An address can be stored in pointer variable.

```
*ted = 25;
beth = *ted;
```



Using a pointer we can directly access the value stored in the variable which it points to.

To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

More at: http://www.cplusplus.com/doc/tutorial/pointers/

Pointers. Declaration and Initialization

```
// declaration and initialisation
float f;
float fArray[100];
float* a = &f;                  // with an address of a float variable
float* a(&f);                   // with an address of a float variable
float* b = a;                   // with a value of another pointer
float* a = fArray;              // with a name of an array. fArray is also a pointer
float* a = (float*)0xB8000000;  // with an explicit address of a memory
float* a = 0;                   // with a zero value
float* a = new float;           // with a memory allocation
```

```
// pointers with constness
float* a;                       // a pointer to float
const float* ca;                // a pointer to const float
float* const ac;                // a constant pointer to float
const float* const cac;         // a constant pointer to const float
```

Pointers. Manipulation with the Memory

```
// allocation of the memory
float*          myFloatPointer = new float;              //a single float
float*          myArrayPointer = new float[100];         //array of floats
MyClassType* myClassPointer = new MyClassType(10,20); //a single object of a class
float*          myFloatAlignedPointer = (float*) _mm_malloc(4,16); //a single float aligned on 16 bytes

// free the memory
delete    myClassPointer;
delete[] myArrayPointer;
delete    myFloatPointer;
_mm_free(myFloatAlignedPointer); // only when allocated with _mm_malloc()



// manipulations with a pointers
*myFloatPointer = 2.f;                  // manipulation with the value pointed by
*(myArrayPointer+1) *= 2.f;             // manipulation with the second element of the array
myArrayPointer[1] *= 2.f;               // equivalent manipulation with the second element
(*myClassPointer).MyPrintFunction();    // call of the class member function
myClassPointer->MyPrintFunction();      // equivalent call of the function

float myFloat = 10;
*myFloatPointer = myFloat ;             // set the value to the cell pointed by the pointer
myFloatPointer = &myFloat;              // store the address of a float to the pointer
```


Functions. Call by Value, by Reference or by Pointer

- Call by value: the value of a variable is copied to a local variable.
  - int increment(int i) { return i++; }
  - int increment(int i, const int a) { return i+a; }
- Call by reference: faster, since the memory for a local variable is not allocated, the function can modify the value of an input variable.
  - void increment(int &i) { i++; }
  - void increment(int &i, const int &a) { i+=a; }
- Call by pointer: the memory for variable is not allocated, the pointer to variable is copied to a local pointer variable, value of a variable can be modified, the array can be given as an input.
  - void increment(int *i) { (*i)++; }
  - void increment(int *i, const int *a) { *i += *a; }
  - void increment(int *array, const int N) {
    for(int i=0; i<N; i++)
    array[i]++;
    }

More at: http://www.cplusplus.com/doc/tutorial/functions2/

Classes in C++

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
    function1();
  access_specifier_2:
    member2;
    function2();
  ...
  } object_names;

class_name::function1() {
  // function1 definition
}
  ...
```

More at: http://www.cplusplus.com/doc/tutorial/classes/

Example of a Class

```
// class example
#include <iostream>
using namespace std;

class CRectangle {
  public:             // the following members can be accessed from outside of the class
    CRectangle(int a, int b): x(a), y(b) {}; // class constructor, equivalent to SetNewValues
    void SetNewValues (int, int);
    int Area () { return (x*y); }
  private:            // the following members can be accessed only by class member functions
    int x, y;
};

void CRectangle::SetNewValues (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect (1,1);
  cout << "area1: " << rect.Area();
  rect.SetNewValues (3,4);
  cout << "area2: " << rect.Area();
  return 0;
}
```

Templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;
template <typename identifier> function_declaration;

To use this function template we use the following format for the function call:

function_name <type> (parameters);

We also have the possibility to write class templates, so that a class can have members that use template parameters as types.

More at: http://www.cplusplus.com/doc/tutorial/templates/

Templates. Function Example

```
 // function template
#include <iostream>
using namespace std;

template <typename T>
T GetMax (T a, T b) { // define
  T result;
  if (a>b)   result = a;
  else       result = b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double l=10.2, m=5e10, n;
  k=GetMax<int>(i, j);  // use
  n=GetMax<double>(l, m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Output:
6
5e10

Templates. Class Example

```
 // class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {  // define class
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax () // define class member function
{
  T result;
  if (a>b)  result = a;
  else        result = b;
  return result;
}

int main () {
  mypair <int> myobject (100, 75); // use
  cout << myobject.getmax();
  return 0;
}
```

C++ also has the possibility to write class templates, so that a class can have members that use template parameters as types.

Output:
100

Templates. Example with Several Parameters

```
#include <iostream>
using namespace std;

template <class T, class T2, int N>
struct myData {  // define struct
    T a;
    T2 b[N];
};

int main () {
  myData <int,int,10> myStruct1; // use
  myData <int,float,20> myStruct2;
  myStruct1.b[5] = 5.2;
  myStruct2.b[9] = 9.1;
  cout << myStruct1.b[5] << endl;
  cout << myStruct2.b[9] << endl;
  return 0;
}
```

Several template parameters can be used

Output:
5
9.1

Exercises

It is recommended to perform exercises within tested environment provided by linux virtual machine:
http://web-docs.gsi.de/~mzyzak/TSSuse.zip

using VirtualBox software:
https://www.virtualbox.org

C++ exercises are cover:

1. Basics (compilation, output to screen, conditions and loops) - 2 exercises.

2. Pointers (memory management, manipulation with pointers, typical mistakes) - 6 exercises.

3. Templates (generalisation of algorithm implementation) - 2 exercises.

Download link: https://www.dropbox.com/sh/eas4kmqbrfk7xrw/s2HHiSnVin

1. Unix Shell and C++ Introduction

Exercises are located: Exercises/Exercise/Exercises/1_CPP/
Solutions: Exercise/Exercises/1_CPP/Solutions/
To compile and run exercise programs use "g++ FILENAME.cpp -o a.out; ./a.out".
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

Unix Shell Introduction

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems (Linux, OSX). There are different shell categories, for example Bourne shell (sh), Bourne-Again shell (bash), C shell (csh), TENEX C shell (tcsh). We consider bash commands, bash is often default shell, or can be called from other shells with '**bash**' command.

The basic bash commands:
pwd — shows you current directory. The current directory is the directory you work in, the most bash commands is applied to it. When you start shell you always start out in your 'home directory'.
cd [dirname] — change current directory. You can get back to your 'home directory by typing '**cd**' without arguments. '**cd ..**' will get you one level up from your current position. You always can add path to **[dirname]**, so you don't have to walk along step by step - you can go to required directory directly, like '**cd ../../dirname1/dirname2**'.
mkdir [dirname] — create a new directory in current directory.
ls [dirname] — list of files in the directory. You can skip the argument, '**ls**' command will shows list of files in the current directory.
ls -l — list your files in 'long format', which contains additional information about each file, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
ls -a — lists all files, including hidden files (the ones whose filenames begin in a dot).
mv [fileOrDirName] [dirname] — moves a file or a directory to another directory.
mv [fileOrDirNameOld] [fileOrDirNameNew] — renames a file or a directory.
cp [filenameOld] [filenameNew] — copies a file.
cp -r [dirnameOld] [dirnameNew] — copies a directory and its content.
rm [filename] — removes a file.
rm -r [dirname] — removes a directory and its content.
grep [string] [filenames] — looks for the string in the files.
wget [fileurl] [dirname] — downloads file from web to the directory.
ssh [user]@[hostname] — connect into a remote machine.
scp [-r] [userOld]@[hostnameOld]:[fileOrDirNameOld] [userNew]@[hostnameNew]:[fileOrDirNameNew] — copies a file or a directory and its content.
g++ [filenames].cpp [flags] -o [executablename] — compile C++ source code with GNU compiler and create executable.
./[executablename] — run executable file.

The shell commands can be collected into script text files. Shell scripts have extension '**.sh**'. Use '**. [bashScriptName].sh**' to execute bash script.

C++ Introduction

1_CPP/1_HelloWorld: description

The first our exercise is a standard program for the beginners in any programming language. The program prints as a result on a screen "Hello World!" message. It is a simple program, but it contains the fundamental components of every C++ program:

Part of the source code of 1_HelloWorld.cpp

```
1  // Run and understand it.
2
3
4  #include <iostream>
5  using namespace std;
6
7  int main() {
8
9    cout << " Hello world " << endl;
10
11   return 0;
12 }
```

Typical output

```
 Hello world
```

comment line

// Run and understand it.

The lines which start with two slashes (*//*) are comments in C++ and do have no influence on the behaviour of the program. The programmer can use it to write brief comments or observations inside the source code.

directives

#include <iostream>

A hash sign (#) starting lines are preprocessor directives. They are no a regular code, but commands for the compiler's preprocessor. The directive *#include <iostream>* will include the *iostream* standard file. The file contains the basic standard input-output library declarations in C++. We will use them later.

using namespace std;

There is a concept of namespace in C++. In general, a **namespace** is a container for a set of identifiers. Namespaces provide a level of indirection to specific identifiers, thus making it possible to distinguish between identifiers with the same exact name, but from different namespaces. For example, the entire C++ standard library is defined within `namespace std`. So in order to access standard functionality we declare with the expression **using namespace std** that we will be using this namespace.

main function

int main ()

Every C++ language program must contain a main() function. It's a core and starting point of every program. Regardless of where it is located in the code the execution will be started with the main function.

The name of the function - main is followed in the code by a pair of brackets(), which optionally can enclose a list of input parameters for the function. For example with *argc* (argument count) and *argv* (argument vector) one can get the number and the values of passed arguments when the application is launched:

int main (int argc, char argv)**

The body of the main function is inside next braces {}. They contain everything program will do during execution.

statement

cout << "Hello World!";

cout is the standard output stream in C++. This statement inserts the line **"Hello World!"** into the output stream (which in our case is the screen).

The *cout operator* is declared in the *iostream* standard file in the *namespace std.* Since we already declared that we are using this namespace, cout operator is available.

Notice that the statement should end with a semicolon (;). One should not forget to put it in the end of every statement, since it's one of the most common syntax errors.

return statement

return 0;

The return statement causes the function to quit. Return has to throw a value in C++, which is called error code. A zero code is generally interpreted as no errors during execution signal. This is the most common way to end a C++ program.

data types

The data types in C++ can be divided in several groups:

- **character types:** can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character.
- **numerical integers:** can store a whole number value, such as 9 or 1024. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values.
- **floating-point numbers:** can represent real values, such as 3.14 or 2.71828, with different levels of precision.
- **booleans:** can only represent one of two states, true or false.

Type	Values	Size (bytes)
bool	“true” and “false”	1
unsigned char	0 … 255	1
signed char	–128 … 127	1
unsigned short int	0 … 65 535	2
short	–32 768 … 32 767	2
unsigned int	0 … 4 294 967 295	4
int	–2 147 483 648 … 2 147 483 647	4
float	3.4e–38 … 3.4e+38	4
double	1.7e–308 … 1.7e+308	8
long double	3.4e–4932 … 3.4e+4932	10 (12,16)

1_CPP/2_ForLoop: description

The next exercise has the same basics elements discussed in previous one, but also it includes an important and frequently used programmer tool: a loop. Loops are useful when one needs to perform an action a certain number of times or while a certain condition is fulfilled.

Part of the source code of 2_ForLoop.cpp

```
1  // Run and understand
2  // countdown using a for loop
3  #include <iostream>
4  using namespace std;
5  int main ()
6  {
7    for (int n=10; n>0; n--) {
8      cout << n << ", ";
9    }
```

```
Part of the source code of 2_ForLoop.cpp
10    cout << "FIRE!\n";
11    return 0;
12  }
```

```
Typical output
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

For the **"for"** loop format is:

for (initialisation; condition; counter change) statement;
and it is intended to repeat the **statement** while the **condition** remains true. This loop is specially designed to perform an action with a counter which may be changed on each iteration.

It works as follows:
1. optionally initialisation is executed. This stage performed only once at maximum.
2. condition is checked. If it is true the loop proceeds, otherwise the loop ends and statement is skipped.
3. statement is executed. It can be either a single statement or a statement block in braces { }. It may also include some operation with a counter.
4. whatever is specified in the change field is done and the loop gets back to step 2.

The initialisation and counter change fields are optional. They can be blank, but in all cases the semicolon signs between them must be present. For example one can write: **for (;n<100;)** or **for (;n<10;n++)** in case when variables were initialised before.
Also there is a possibility to use more then one counter. In this case one should put a comma operator **(,)** in-between. This operator works as an expression separator in case where only one is generally expected:

```
for ( n=0, i=50 ; n!=i ; n++, i-- )
{ }
```

This loop will execute for 25 times if neither n or i are modified in the loop body.

Pointers.
Any variables in C++ can be addressed with its' identifier (name). This is the easiest way when we don't need to care about the physical location of our data in the memory. However, there is the second way to address a variable, directly using its location in the memory.
The computer memory can be represented as a sequence of memory cells of one byte each. These cells are numbered in a consecutive way. Each cell has a unique number in the whole available memory. Every next cell has the number of the previous one plus one. This way we can claim that the cell number 55 definitely follows the cell number 54.
As soon as we declare a variable, the amount it needs in memory is allocated in a specific location (memory address). Operating system performs this task automatically during runtime. This memory address locates a variable in the memory and is called a reference to that variable. This reference can be obtained by adding an ampersand sign (&), so-called **reference operator**, in front of the identifier. One can read it as "get address of".

```
float a; // declaring a float a
&a; // getting address of a
```

A variable which stores such memory address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store. All pointers a have a special pinter type. While declaring a pointer one has to specify this type. The pointer type is obtained by adding an asterisk after the type it points to. The declaration of pointers has such a format:

type * name;

where **type** is the data type of the value that the pointer points to. For example:

```
float* a; // a pointer to float
const float* ca; // a pointer to const float
```

Although the data to which different pointers point to doesn't need the same amount of space in the memory, all pointer types occupy the same amount of memory (this amount may vary from platform to platform).

Using a pointer we can directly access the variable stored in the object which it points to by adding an asterisk sing (*), which is called a **dereference operator**. It should not be confused with a pointer type asterisk, since we just use the same sing for different operation. Here one can compere two approaches to access object by identifier and by pointer to perform the same action - increment:

```
int one = 1; // declaring an integer of value 1
one ++; // increment
```

```
int one = 1; // declaring an integer of value 1
int* pointer = & one; //declaring a pointer to int
(*pointer)++; // increment
```

Arrays

An array is a number of elements with the same type placed in one memory location. These elements can be individually addressed by adding an consecutive number to a unique identifier. In order to declare a regular array **name_of_array** of **N** elements with type **type** one has to do it the same way one declares a single element with an addition of number of elements in braces. For example:

```
type name_of_array [N] ;
```

When declaring an array this way its elements will not be initialised with any default value, until we store some value in them. there is a possibility to assign initial values to each element by enclosing the values in braces { }:

```
int nicely_initialised_array [5] = { 16, 2, 77, 40, 12071 };
```

One can create multidimensional arrays. They can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a bi-dimensional table made of elements.

The concept of array is bind to the concept of pointer in C++. In fact, the identifier of an array is equivalent to the address of its first element.

Dynamic memory

Until now we only allocate memory for our variables having the size of memory we needed in advance before the runtime. For this we used static memory. However sometimes this amount of memory needed can only be determined during execution of the program. For example, in the case when we wait for user input. C++ gives the opportunity to implement program in this case. This tool is called dynamic memory and can be used with the help of operators **new** and **delete.**

New is the operator to request for dynamic memory. It should be followed by a data type and, optionally for an array, the number of elements in the brackets [ ]. This operator returns a pointer to the beginning of the allocated block of memory. This should be done in a form:

```
pointer = new type;
pointer = new type [number_of_elements]
```

The first expression is for allocating memory for a single element of type **type**. The second one is used to assign an array of elements with type **type**, where **number_of_elements** is an integer number.

```
int * array_of_integers; // pointer to integer
array_of_integers = new int [10]; // array of 10 integers, array_of_integers points to the 1st
```

Since the amount of available memory is limited, once dynamic memory is no longer needed it should be freed, so that the memory is available again. It's not done automatically like in case of static memory. In this case we have to use **delete** operator in this way:

delete pointer;
delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for an arrays.

Allocating aligned memory blocks

Aligned memory we will need while working with SIMD instructions in the future. To allocate and free aligned blocks of memory use the **_mm_malloc** and **_mm_free** intrinsics. These intrinsics are based on malloc and free, which are in the libirc.a library. The syntax for these intrinsics is as follows:

void* _mm_malloc (size_t size, size_t align)
void _mm_free (void *p)

The **_mm_malloc** routine takes an extra parameter, which is the alignment constraint. This constraint in bytes must be a power of two. The pointer that is returned from **_mm_malloc** is guaranteed to be aligned on the specified boundary. Here is an example of aligned memory allocation:

float* array = (float*) _mm_malloc(sizeof(float)*10, 4); // array of floats aligned on 4

1_CPP/3_MemoryAllocation: description

In this pointer exercise one is supposed to allocate memory for an array in 3 different ways:

1) statically for known size of array;
2) dynamically for unknown size not aligned;
3) dynamically for unknown size aligned.

Part of the source code of 3_MemoryAllocation.cpp

```
1  #include”xmmintrin.h” // for _mm_malloc
2  #include<iostream>
3  #include<cmath>// for sin function
4  using namespace std;
5  int main ( int argc, char** argv )
6  {
7    // When the size of the array is known
8    const int SIZE = 10;
9    float Array_static[SIZE];
10   int i;
11   for ( i = 0; i < SIZE; ++i)
12   {
13     Array_static[i] = sin(i) * 10.f + i ;
14   }
15   // When the size is unknown
16   int size;
17   cout << "Size of dynamic: ";
18   cin >> size; cout << endl;
19   float* Array_dynamic = new float[size];
20   // TODO fill Array_dynamic with sin(i) * 10.f + i;
21   // Print the Array_static array to the screen
22   std::cout << "Array_static ";
23   for(i=0; i<SIZE; ++i)
24     cout << Array_static[i] << " ";
25   cout << endl;
```

Part of the source code of 3_MemoryAllocation.cpp

```
27    // Print the Array_dynamic array to the screen
29    std::cout << "Array_dynamic ";
30    // TODO print to screen Array_dynamic elements
31    // first element
32    cout << endl << "* Array_dynamic:   " << * Array_dynamic << endl;
33    * Array_dynamic = 42.3f;
34    cout << "* Array_dynamic:   " << * Array_dynamic << endl;
35    // And the * Array_dynamic realy points to the first element:
36    cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl;
37    delete[] Array_dynamic;
38    return 0;
39  }
```

Typical output

```
Size of Array_dynamic and Array_dynamic_aligned: 5

Array_static 0 9.41471 11.093 4.4112 -3.56802 -4.58924 3.20585 13.5699 17.8936
13.1212
Array_dynamic Array_dynamic_aligned
* Array_dynamic:   0
* Array_dynamic:   42.3
Array_dynamic[0]: 42.3
Position in memory:
0x7fff8787f3a0
0x23cf010
0x23d0000
```

1_CPP/3_MemoryAllocation: solution

We allocate memory as it is asked:
1) statically for known size of array:
const int SIZE = 10;
float Array_static[SIZE];
2) dynamically for unknown size not aligned:
float* Array_dynamic = new float[size];
3) dynamically for unknown size aligned:
float* Array_dynamic_aligned = (float*) _mm_malloc(sizeof(float)*size, 16*16*16);
In each case we can access needed array element the same manner:
array_name[num_element] in order to print it.
In case of static array we don't need to free memory afterwards since it is freed automatically, but in both dynamic cases we have to do it using corresponding delete operator:
2) **delete[] Array_dynamic;**
3) **_mm_free(Array_dynamic_aligned);**

In the end of exercise we perform some action with array identifier in order to better understand the concept of array: identifier g_koeff is a pointer, storing address of the 1st element of array g_koeff, dereferenced *g_koeff — is a 1st element itself.

cout << endl << "* Array_dynamic: " << * Array_dynamic << endl; // print 1st element value
*** Array_dynamic = 42.3f;** // store 42.f to 1st element value
cout << "* Array_dynamic: " << * Array_dynamic << endl; // print 1st element value
cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl; // other way to print it

Part of the source code of 3_MemoryAllocation_solution.cpp

```
1  #include”xmmintrin.h” // for _mm_malloc
2  #include<iostream>
3  #include<cmath>// for sin function
4  using namespace std;
5  int main ( int argc, char** argv )
6  {
7    // When the size of the array is known
8    const int SIZE = 10;
9    float Array_static[SIZE];
10   int i;
11   for ( i = 0; i < SIZE; ++i)
12   {
13     Array_static[i] = sin(i) * 10.f + i ;
14   }
15
16   // When the size is unknown
17   int size;
18   cout << "Size of dynamic and dynamic_aligned: ";
19   cin >> size; cout << endl;
20
21   float* Array_dynamic = new float[size];
22   // The access to the elements of this array is the same as for the
     usuall array.
23   for ( i = 0; i < size; ++i )
24   {
25     Array_dynamic[i] = sin(i) * 10.f + i;
27   }
29
30   float* Array_dynamic_aligned = (float*) _mm_malloc(sizeof(float)*size,
     16*16*16);
31   for ( i = 0; i < size; ++i )
32   {
33     Array_dynamic_aligned[i] = sin(i) * 10.f + i;
34   }
35
36   // Print the f_koeff array to the screen
37   std::cout << "Array_static ";
38   for(i=0; i<SIZE; ++i)
39
40     cout << Array_static[i] << " ";
41   cout << endl;
42
43   // Print the g_koeff array to the screen
44   std::cout << "Array_dynamic ";
45   for(float *p = Array_dynamic; p < Array_dynamic + size; ++p)
46     cout << *p << " ";
47   cout << endl;
48
49     // Print the Array_dynamic_aligned array to the screen
```

Part of the source code of 3_MemoryAllocation_solution.cpp

```
50    std::cout << "Array_dynamic_aligned ";
51    for(i=0; i<size; ++i)
52   cout << Array_dynamic_aligned[i] << " ";
53    cout << endl;
54
55
56    //
57    cout << endl << "*Array_dynamic:   " << *Array_dynamic << endl;
58    *Array_dynamic = 42.3f;
59    cout << "*Array_dynamic:   " << *Array_dynamic << endl;
60    // And the *Array_dynamic really points to the first element:
61    cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl;
62
63    cout << "Position in memory: " << endl << Array_static << endl <<
64  Array_dynamic << endl << Array_dynamic << endl;
65    delete[] Array_dynamic;
66    _mm_free(Array_dynamic);
67    return 0;
68  }
69
```

Typical output

```
Size of Array_dynamic and Array_dynamic_aligned: 5

Array_static 0 9.41471 11.093 4.4112 -3.56802 -4.58924 3.20585 13.5699 17.8936
13.1212
Array_dynamic 0 9.41471 11.093 4.4112 -3.56802
Array_dynamic_aligned 0 9.41471 11.093 4.4112 -3.56802

* Array_dynamic:   0
* Array_dynamic:   42.3
Array_dynamic[0]: 42.3
Position in memory:
0x7fffc6f6a1f0
0x138c010
0x138d000
```

1_CPP/4_NewDeleteOperators: description

In the next pointer exercise one is supposed to find bugs in different pointer usage situations:

Part of the source code of 4_NewDeleteOperators.cpp

```
...
11  const short k = 1;
12
13  int main ( )
14  {
15    float* p = new float(123); // allocate the memory, initialize with 123
16    float* p1 = p;
17    p1++;
18
19    cout << " PART 1 " << endl;
```

Part of the source code of 4_NewDeleteOperators.cpp

```
20    cout << "Initial value:        " << p << "  " << *p << endl;
21    (*p)++; //increase the value
22    cout << "Increased value:      " << p << "  " << *p << endl;
23
24      //clean the memory;
25    cout << " PART 2 " << endl;
26    cout << "Address p before delete: " << p << "  " << *p << endl;
27    delete p;
28    cout << "Address p after delete:  " << p << "  " << *p << endl;
29
30    if(k == 2) {
31      cout << " PART 3 " << endl;
32      cout << "Address p before delete: " << p << "  " << *p << endl;
33      delete p; //free the memory
34      cout << "Address p after delete:  " << p << "  " << *p << endl;
35    }
36    if(k >= 2) {
37      cout << " PART 4 " << endl;
38      cout << "Address p1 before delete: " << p1 << "  " << *p1 << endl;
39      delete p1; //free the memory
40      cout << "Address p1 after delete:  " << p1 << "  " << *p1 << endl;
41    }
…
```

Typical output with k = 1

```
 PART 1
Initial value:        0xa3b010  123
Increased value:      0xa3b010  124
 PART 2
Address p before delete: 0xa3b010  124
Address p after delete:  0xa3b010  0
```

Part of typical output with k = 2

```
 PART 3
Address p before delete: 0x1f8d010  0
*** glibc detected *** ./a.out: double free or corruption (fasttop):
0x0000000001f8d010 ***
```

Part of typical output with k = 3

```
 PART 4
Address p1 before delete: 0xbec014  0


*** glibc detected *** ./a.out: free(): invalid pointer: 0x0000000000bec014 ***
```

1_CPP/4_NewDeleteOperators: solution

While being run this program receives *"double free or corruption"* message. Let us understand why.

In this program the main function begins by declaring a pointer, which points to dynamically allocated float initialised with the value of 123.

Right after that, we introduce one more pointer p1, which points the same float. In the line 10 we increment the pointer p1. Note that after this operation p1 now points to the next float in the memory. However the actual value of the allocated float is not changed and still equals to 123.

In lines 12-14 we print the address of the float *(p)* and its value *(*p),* which is still 123, increase the value of *float ((*p)++)* with the use of dereferencing operator and print the address, which remains the same, and the increased value (124). The outcome of these lines:

Initial value: 0x602010 123
Increased value: 0x602010 124

In lines 17-18 we free the memory to which p was pointing. Since the float was allocated dynamically we use delete operator with usual syntax by passing the pointer to memory block previously allocated with new operator: **delete p;**

That explains why after this operation for printing address and the value of float we get:
0x602010 0.

So address remains the same. However, the value is set to 0 by delete operator.

In the line 23 we are trying to free dynamically allocated memory block the 2nd time, which is not allowed in c++. This produce “double free or corruption“ message. The easiest way to fix this bug is to change condition in the line 21 to: **if(0),**

so that lines 22-24 are never executed.

In the line 23 we are trying to free memory block, to which points pointer p1. As it was mentioned before, after executing the line 10 it points to the memory block, which was not allocated. This is not allowed in C++. We can fix this bug the same manner, by commenting these lines with the use of condition clause **if(0)** as it was done previously.

Part of the source code of 4_NewDeleteOperators_solution.cpp

```
...
11  const short k = 1;
12
13  #include <iostream>
14  using namespace std;
15
16  int main ( )
17  {
18    float* p = new float(123); // allocate the memory, initialize with 123
19    float* p1 = p;
20    p1++;
21
22    cout << " PART 1 " << endl;
23    cout << "Initial value:        " << p << "  " << *p << endl;
24    (*p)++; //increase the value
25    cout << "Increased value:      " << p << "  " << *p << endl;
26
27      //clean the memory;
28    cout << " PART 2 " << endl;
29    cout << "Address p before delete: " << p << "  " << *p << endl;
30    delete p;
31    cout << "Address p after delete:  " << p // 1st case
32         << "  " << *p << endl; // 2nd
33
34    if(k == 2) {
35      cout << " PART 3 " << endl;
```

Part of the source code of 4_NewDeleteOperators_solution.cpp

```
36      cout << "Address p before delete: " << p << "  " << *p << endl; // 3rd &
        4th
37      delete p; //free the memory // 5th
38      cout << "Address p after delete:  " << p << "  " << *p << endl; // 6th &
        7th
39    }
40    if(k >= 2) {
41      cout << " PART 4 " << endl;
42      cout << "Address p1 before delete: " << p1 << "  " << *p1 << endl; // 8th &
        9th
43      delete p1; //free the memory // 10th
44      cout << "Address p1 after delete:  " << p1 << "  " << *p1 << endl; // 11th
        & 12th
45    }
46
47    return 0;
48  }
```

1_CPP/5_PointersAndFunctions: description

In the next exercise one is supposed to find two bugs, which leads to segmentation fault.

Part of the source code of 5_PointersAndFunctions.cpp

```
11  void piPointer1(float* pi) {
12    *pi = 3.14;
13  }
14
15  float* piPointer2() {
16    float pi = 3.1415;
17    return &pi;
18  }
19
20  int main() {
21    float* pi1;
22    piPointer1(pi1);
23    cout << *pi1 << endl;
24
25    float* pi2 = piPointer2();
26    cout << *pi2 << endl;
27    delete pi2;
28
29    return 0;
30  }
```

Typical output

```
Segmentation fault
```

1_CPP/5_PointersAndFunctions: solution

In order to examine this code first of all remember that C++ program always begins with execution of main function.

The main function in this case starts with introducing a pointer *“pi2”,* which points nowhere up to now. This pointer is given to the function **piPointer1**, which tries to save a value at the place it points to. This would fail. To fix it one needs to allocate memory for pi1 variable and free it after use.

Function **piPointer2** does not need an argument, but it returns pointer to local variable. This variable allocated and destroyed inside of the function, therefore one can not use it outside of the function. Even more - one can not free the memory for this variable, it is already freed. The solution would be to allocate memory dynamically.

Part of the source code of 5_PointersAndFunctions_solution.cpp

```
11 void piPointer1(float* pi) {
12   *pi = 3.14;
13 }
14
15 float* piPointer2() {
16   float* pi = new float;
17   *pi = 3.1415;
18   return pi;
19 }
20
21 int main() {
22   float* pi1 = new float;
23   piPointer1(pi1);
24   cout << *pi1 << endl;
25   delete pi1;
26
27   float* pi2 = piPointer2();
28   cout << *pi2 << endl;
29   delete pi2;
30
31   return 0;
32 }
```

Typical output

```
3.14
3.1415
```

1_CPP/6_Factorial: description

The next program is intended to calculate the factorials of N natural numbers. One is supposed to understand the code and find a bug:

Part of the source code of 6_Factorial.cpp

```
1  // Will this program work? Find a bug.
2
3  #include<iostream>
4  using namespace std;
5
6  const int N = 10;
7
8    // Get set of the factorials of the first N numbers
9  int *GetFactorials(){
10   int a[N];
11
```

Part of the source code of 6_Factorial.cpp

```
12    a[0] = 1;
13    for( int i = 1; i < N; ++i )
14      a[i] = i*a[i-1];
15
16    return a;
17  }
18
19  int main() {
20      // Get set of the factorials of the first N numbers
21    int *a = GetFactorials();
22
23      // print it
24    for( int i = 0; i < N; ++i )
25      cout << a[i] << endl;
27
29    return 0;
30  }
```

Part of output

```
1
51
-1190761318
51
6295008
0
-1136061198
51
1653124016
32767
*** glibc detected *** ./a.out: munmap_chunk(): invalid pointer:
0x00007fff6288aa80 ***
```

1_CPP/6_Factorial: solution

In order to understand the problem with this program lets first consider GetFactorials function. The line 10 shows that the function returns pointer to the integer:

int *GetFactorials()

In the function body we introduce an array of integers with the size of N:

int a[N];

In the line 14 we have a for loop, which computes factorial values for 1st N natural numbers and stores the results in a static **array a[N].**

In the end function returns the pointer to the first array element *a*.

Already here we face a problem, since **a[N]**, being a static array, has a scope limited to the function body only. So we it makes no sense to return pointer to it, because the memory is freed automatically after the function execution is finished. In order to fix this bug we should replace line 11 with:

int *a = new int[N];

So that the memory is allocated dynamically and will only be freed after calling delete operator in the end of the program in main function:

delete[ ] a;

Part of the source code of 6_Factorial_solution.cpp

```
1  // Will this program work? Find a bug.
2
3  #include<iostream>
4  using namespace std;
5
6  const int N = 10;
7
8    // Get set of the factorials of the first N numbers
9  int *GetFactorials(){
10   int *a = new int[N];
11
12   a[0] = 1;
13   for( int i = 1; i < N; ++i )
14     a[i] = i*a[i-1];
15
16   return a;
17 }
18
19 int main() {
20     // Get set of the factorials of the first N numbers
21   int *a = GetFactorials();
22
23     // print it
24   for( int i = 0; i < N; ++i )
25     cout << a[i] << endl;
27  int *a = new int[N];
29   delete[] a;
30   return 0;
31 }
```

Typical output

```
1
1
2
6
24
120
720
5040
40320
362880
```

1_CPP/7_FunctionArgument: description

In the next pointer exercise one is supposed to write different version of increment function:

Part of the source code of 7_FunctionArgument.cpp

```
11  void increase1(int arg)
12  {
13    arg++;
14  }
15
16  int increase2(int arg)
17  {
18    arg++;
19    return arg;
20  }
21
22  void pointer_increase(int* arg)
23  {
24    // TODO
25  }
26
27  void reference_increase(int& arg)
28  {
29    // TODO
30  }
31
32  int main ()
33  {
34    int number = 0;
35    cout << "Number is: " << number  << endl;
36    increase1( number ); // Has no effect.
37    cout << "Number is: " << number  << endl;
38    number = increase2( number );  // increase number by 1.
39    cout << "Number is: " << number << endl;
40    pointer_increase( /* TODO */ ); // increase number by 1.
41    cout << "Number is: " << number << endl;
42    reference_increase( /* TODO */ ); // increase number by 1.
43    cout << "Number is: " << number << endl;
44    return 0;
45  }
```

Typical output

```
exPointers4.cpp: In function 'int main()':
exPointers4.cpp:22: error: too few arguments to function 'void
pointer_increase(int*)'
exPointers4.cpp:40: error: at this point in file
exPointers4.cpp:27: error: too few arguments to function 'void
reference_increase(int&)'
exPointers4.cpp:42: error: at this point in file
```

1_CPP/7_FunctionArgument: solution

Functions should differ from each other in a way of passing argument. In the main body we introduce an integer number with value of 0. After this we increase its value by 4 different ways and print the result after each time in the lines 28-36.

Now lets have a closer look into function **increase1**. In this case we pass number by value, however the function is void type, so it doesn't return anything, as one can see it in line 4:

void increase1(int arg)

In this case we create a local copy of variable “number” in function, increase it in line 6:

arg++;

However, this action doesn't change the global variable “number” in the main body. In case of **increase2** we pass argument by value, but in this case the function type is not void, but integer, as it is declared:

int increase2(int arg)

So it returns the copy of locally increased variable “number”.
Thus, after line 31:

number = increase2(number); // increase number by 1.

value of global variable “number” will be increased by 1.
In the case of **pointer_increase** the argument is passed by pointer:

void pointer_increase(int* arg)

So we don't create a local copy of variable in this case, in order to increase the value of global variable, we should use dereference operator in this manner:

(*arg)++;

In the case of **reference_increase** the argument is passed by reference:

void reference_increase(int& arg)

In this case every action we perform with function argument automatically will be done with global variable.

So in this case we can perform increment as easy that:

arg++;

Part of the source code of 7_FunctionArgument_solution.cpp

```
22 void pointer_increase(int* arg)
23 {
24   (*arg)++;
25 }
26
27 void reference_increase(int& arg)
28 {
29   arg++;
30 }
```

Typical output

```
Number is: 0
```

Typical output

```
Number is: 0
Number is: 1
Number is: 2
Number is: 3
```

1_CPP/8_Arrays: description

The task for this exercise is to explain the output.

Part of the source code of 8_Arrays.cpp

```
11  int main()
12  {
13    const int N = 10;
14
15      // declare pointers
16    int* p1, *p2;
17
18      // allocate memory
19    p1 = new int[N];
20    p2 = new int[N];
21
22      // fill
23    for( int i = 0; i < N; ++i ) {
24      p1[i] = i;
25      p2[i] = i;
26    }
27
28      // change some values. Will arrays change?
29    p1[11] = 1011;
30    p1[15] = 1015;
31    p2[13] = 2013;
32
33      // print
34    for( int i = 0; i < N; ++i ) {
35      cout << "p1[" << i << "] = " << p1[i] << " " << "p2[" << i << "] = " <<
    p2[i] << endl;
36    }
37
38    delete[] p1;
39    delete[] p2;
40  }
```

Typical output

```
p1[0] = 0 p2[0] = 0
p1[1] = 1 p2[1] = 1
p1[2] = 2 p2[2] = 2
p1[3] = 3 p2[3] = 1015
p1[4] = 4 p2[4] = 4
p1[5] = 5 p2[5] = 5
```

```
Typical output
p1[6] = 6 p2[6] = 6
p1[7] = 7 p2[7] = 7
p1[8] = 8 p2[8] = 8
p1[9] = 9 p2[9] = 9
*** glibc detected *** ./a.out: free(): invalid next size (fast):
0x0000000002494010 ***
```

1_CPP/8_Arrays: solution

The task for this exercise is to explain the output. To do this, lets try to understand what is done in the main body. In line 6 the const integer "*N*" with value 10 is introduced. In line 9 we declare 2 pointers to integer "*p1*", "*p2"*.

In the lines 12-13 we dynamically allocate memory for 2 arrays with N=10 elements in each.

In the lines 16-20 we have a for loop, which fills both arrays with array element numbers: from 0 to (N-1).

In lines 22-23 we try to change one element of each array, number 15 in array "p1" and 13 in number "*p2*":

```
p1[15] = 1005;
p2[13] = 2003;
```

However, as we learned from lines 12-13 for both arrays we allocate memory only for 10 elements in each. This means that in our case it's not correct to address any elements with number higher than 9.

In order to see the outcome of our action we print arrays elements with the use of loop in the line 27. The outcome of program shows us that we have change the element with number 4 in array p2. This can easily be explained and give us some insight to the process of allocated dynamically. After performing lines:

```
p1 = new int[N];
p2 = new int[N];
```

processor allocates 2 memory blocks each of 10 floats. The blocks are located in the same place in memory: first array "*p1*", "*p2*" follows right after "*p1*".

This is the reason, why addressing non existent element of array *p1* with number *15*, we actually change 6th element of array "*p2*".

With line:

```
p2[13] = 2003;
```

We corrupt some not yet allocated memory. In this case it didn't show segmentation fault, although it's not right way to code. This also shows that even if program produce correct results, it doesn't necessarily mean that it has no bugs :-).

1_CPP/9_Templates: description

Templates

Function templates are special functions that can operate with generic types. This allows to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still- undefined type.

To use this function template we use the format for example for two integers:

```
int x,y;
GetMax <int> (x,y);
```

Part of the source code of 9_Templates.cpp

```
11  template <typename T>
12  T GetMax (T a, T b) {
13    T result;
14    if (a > b)   result = a;
15    else         result = b;
16    return result;
17  }
18
19  int main () {
20    int i=5, j=6;
21    double l=9.2, m=2e9;
22    cout << GetMax<int>(i,j) << endl;
23    cout << GetMax<double>(l,m) << endl;
24    cout << "float: " <<  GetMax<float>(l,m) << endl;
25    cout << "int: "   <<  GetMax<int>(l,m) << endl;
26    cout << "short: " <<  GetMax<short>(l,m) << endl;
27    cout << "char: "  <<  int(GetMax<char>(l,m)) << endl;
28    return 0;
29  }
```

Typical output

```
6
2E+09
float: 2e+09
int: 2000000000
short: 9
char: 9
```

Lets start examining this code with template function GetMax. This function takes 2 parameters (*a, b*) of the same type T by value and returns the result with the same type T, as it is written in the line 7.

In the body we declare a variable "*result*" with type T.

In the lines 6-9 we store the greater variable among (*a, b*) to the value of "result" with the use of conditional clause. In the last line we return the result.

In the main body we declare 2 integers and 2 doubles in lines 13-14.

In lines 15-16 we call our template function with different types. It always returns the greater value.

1_CPP/10_Epsilon: description

In the last exercise one is supposed to estimate the machine error epsilon with some precision for different types: float, double and long double with the use of templates. Machine epsilon can be found as the smallest value of e, such that (1 + e) is not equal to 1.

1_CPP/10_Epsilon: solution

Part of the source code of 10_Epsilon_solution.cpp

```
16 template <class T>
17 T Epsilon() {
18 //  cout << " (Type size: " << sizeof(T) << ") ";
19
20   const T one = 1;
21   T e = one;
22
23   for( T oneP = one + e/2; abs(oneP - one) > 0; oneP = one + e/2 ) {
24     e = e/2;
25   }
26   return e;
27 }
28
29 int main () {
30   cout << "Machine epsilon for float =       ";
31   cout << Epsilon<float>() << endl;
32   cout << "1 + e - 1 = " << 1 + Epsilon<float>() - 1 << endl;
33
34   cout << "Machine epsilon for double =      ";
35   cout << Epsilon<double>() << endl;
36   cout << "1 + e - 1 = " << 1 + Epsilon<double>() - 1 << endl;
37
38   cout << "Machine epsilon for long double = ";
39   cout << Epsilon<long double>() << endl;
40   cout << "1 + e - 1 = " << 1 + Epsilon<long double>() - 1 << endl;
41
42   cout << "Machine epsilon for int =         ";
43   cout << Epsilon<int>() << endl;
44   cout << "1 + e - 1 = " << 1 + Epsilon<int>() - 1 << endl;
45
46   return 0;
47 }
```

Typical output

```
Machine epsilon for float =       1.19209e-07
1 + e - 1 = 1.19209e-07
Machine epsilon for double =      2.22045e-16
1 + e - 1 = 2.22045e-16
Machine epsilon for long double = 1.0842e-19
```

	Typical output
	1 + e - 1 = 1.0842e-19 Machine epsilon for int = 1 1 + e - 1 = 1

Since machine epsilon can be found as the smallest value of e, such that (1 + e) is not equal to 1, we can introduce a for loop, which will stop as soon as condition ((1 + e)=1) is fulfilled. There could be many ways to write this condition, we have done it in this way:

for (T oneP = one + e/2; abs(oneP - one) > 0; oneP = one + e/2) e = e/2;

This way starting from 1 we make e two times smaller each time till the point there condition:

((1 + e)==1)

is fulfilled. As soon as it happens we stop the loop execution and return the final e value. Since the function is written with the use of templates, we can run it for different variable types, in order to find out the precision for each one.

HPC Practical Course
Part 2.1

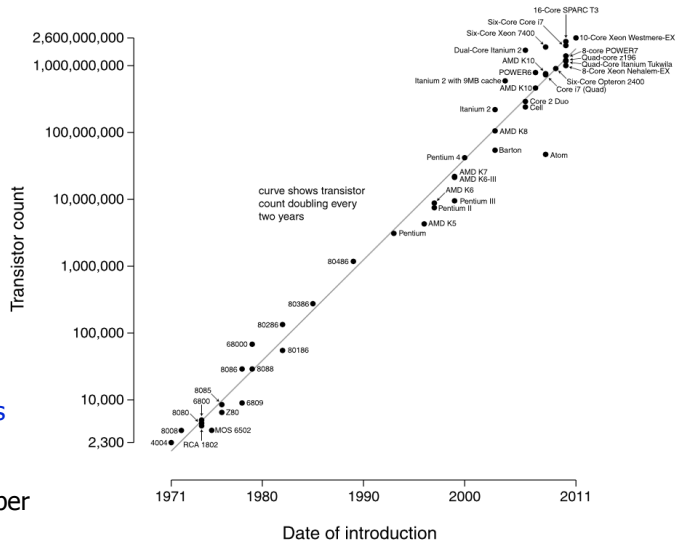
SIMD with headers

V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak

Goethe University of Frankfurt am Main

30 Apr 2014

Moore's Law

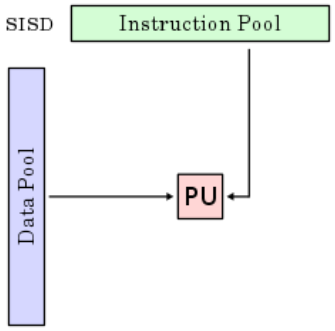


Moore's Law:
- the number of transistors doubles approximately every two years
- surprisingly good prediction
- but it "only" talks about the number of transistors
- the clock rate mostly fixed

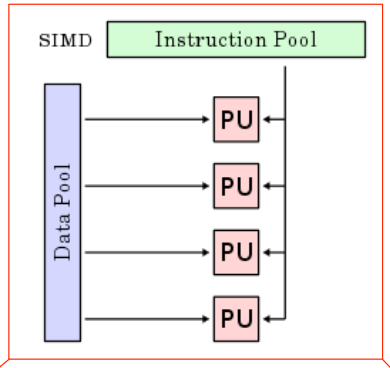
performance increase by means of:
- improved parallelism
- faster and larger memory caches

Computer architectures

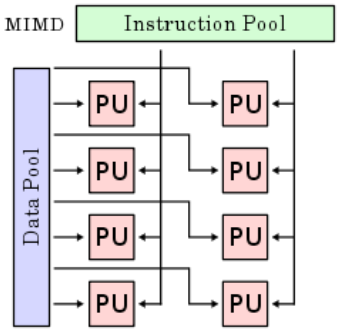
Single Instruction Sindle Data



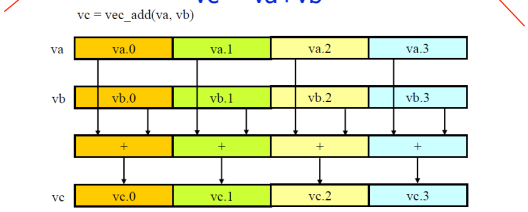
Single Instruction Multiple Data



Multiple Instruction Multiple Data



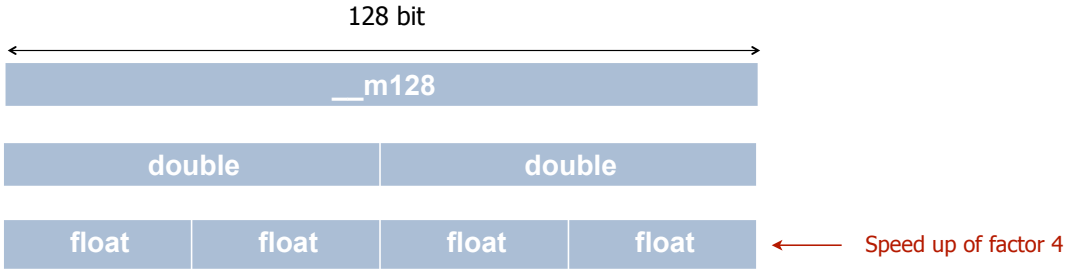
vc = va+vb



Taken from: http://en.wikipedia.org/wiki/Flynn's_taxonomy

SIMD registers

SIMD register:



- Processors with 256 bit and 512 bit registers exist

Basic single precision SSE SIMD instructions

Arithmetic:

_mm_add_ps (Va,Vb)	a + b
_mm_sub_ps (Va,Vb)	a - b
_mm_mul_ps (Va,Vb)	a * b
_mm_div_ps (Va,Vb)	a / b

Logical:

_mm_and_ps (Va,Vb)	a & b
_mm_or_ps (Va,Vb)	a | b
_mm_xor_ps (Va,Vb)	a ^ b

Comparison:

_mm_cmplt_ps (Va,Vb)	a < b
_mm_cmple_ps (Va,Vb)	a <= b
_mm_cmpgt_ps (Va,Vb)	a > b
_mm_cmpge_ps (Va,Vb)	a >= b
_mm_cmpeq_ps (Va,Vb)	a == b

Extra:

_mm_min_ps (Va,Vb)	min (a, b)
_mm_max_ps (Va,Vb)	max (a, b)
_mm_rcp_ps (Va)	1/a
_mm_sqrt_ps (Va)	sqrt(a)
_mm_rsqrt_ps (Va)	1/sqrt(a)

There are much more instructions, including instructions for SIMD vectors with double precision values

Wrapper C++ Header

The instructions can be wrapped by C++ class,
which overloads standard operators (+, -, *, \, >, etc.) for the convenience of user.

Partial source code of P4_F32vec4.h

```
44  class F32vec4
45  {
46   public:
47
48    __m128 v;
49
50    float & operator[]( int i ){ return (reinterpret_cast<float*>(&v))[i]; }
51    float   operator[]( int i ) const { return (reinterpret_cast<const float*>(&v))[i]; }
52
53    F32vec4( ):v(_mm_set_ps1(0)){}
54    F32vec4( const __m128 &a ):v(a) {}
55    F32vec4( const float &a ):v(_mm_set_ps1(a)) {}
56
57    F32vec4( const float &f0, const float &f1, const float &f2, const float
    &f3 ):v(_mm_set_ps(f3,f2,f1,f0)) {}
...
62    /* Arithmetic Operators */
63    friend F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return _mm_add_ps(a,b); }
...
72    /* Square Root */
73    friend F32vec4 sqrt ( const F32vec4 &a ){ return _mm_sqrt_ps (a); }
...
214 } __attribute__ ((aligned(16)));
215
216
217 typedef F32vec4 fvec;
218 typedef float  fscal;
219 const int fvecLen = 4;
```

SIMDization example

Let's vectorize c=a+b

```
float a[1000];
float b[1000];
float c[1000];

 // add
for( int i = 0; i < 1000; i++ ) {
   c[i] = a[i]+b[i];
}
```

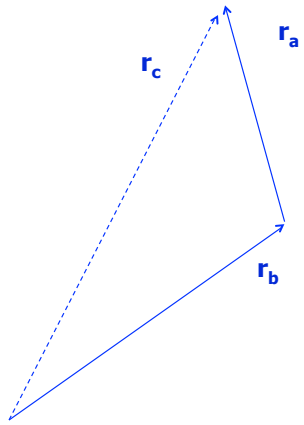
SIMDization

```
float a[1000];
float b[1000];
float c[1000];

fvec aV[250];
fvec bV[250];
fvec cV[250];

 // copy
for( int i = 0; i < 250; i++ ) {
   for( int iv = 0; iv < 4; iv++ ) {
      aV[i][iv] = a[i*4+iv];
      bV[i][iv] = b[i*4+iv];
   }
}

 // add
for( int i = 0; i < 250; i++ ) {
   cV[i] = aV[i]+bV[i];
}

 // copy back
for( int i = 0; i < 250; i++ )
   for( int iv = 0; iv < 4; iv++ )
      c[i*4+iv] = cV[i][iv];
```


Vectorization Approaches: Task

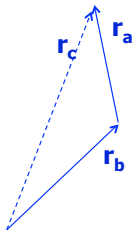
Task:

- vector r = {x, y, z}
- calculate vector sum $r_c = r_a + r_b$
- for many pairs of vectors: r_{a1}, r_{a2}, $r_{a3,\ldots}$; r_{b1}, r_{b2}, $r_{b3,\ldots}$



Vectorization approaches 2

Task:
- vector r = {x, y, z}
- calculate vector sum $r_c = r_a + r_b$
- for many pairs of vectors: $r_{a1}, r_{a2}, r_{a3,...}; r_{b1}, r_{b2}, r_{b3,...}$



Inside the subtask (pair)

x_{a1}	y_{a1}	z_{a1}	_
+			
x_{b1}	y_{b1}	z_{b1}	-
=			
x_{c1}	y_{c1}	z_{c1}	-

x_{a2}	y_{a2}	z_{a2}	_
+			
x_{b2}	y_{b2}	z_{b2}	-
=			
x_{c2}	y_{c2}	z_{c2}	-

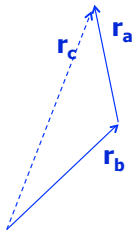
x_{a3}	y_{a3}	z_{a3}	_
+			
x_{b3}	y_{b3}	z_{b3}	-
=			
x_{c3}	y_{c3}	z_{c3}	-

... ...

- Usually is not optimal
- Do not scale

Vectorization approaches 3

Task:
- vector r = {x, y, z}
- calculate vector sum $r_c = r_a + r_b$
- for many pairs of vectors: $r_{a1}, r_{a2}, r_{a3,...}; r_{b1}, r_{b2}, r_{b3,...}$



Between the subtasks

x_{a1}	x_{a2}	x_{a3}	x_{a4}
+			
x_{b1}	x_{b2}	x_{b3}	x_{b4}
=			
x_{c1}	x_{c2}	x_{c3}	x_{c4}

y_{a1}	y_{a2}	y_{a3}	y_{a4}
+			
y_{b1}	y_{b2}	y_{b3}	y_{b4}
=			
y_{c1}	y_{c2}	y_{c3}	y_{c4}

z_{a1}	z_{a2}	z_{a3}	z_{a4}
+			
z_{b1}	z_{b2}	x_{b3}	z_{b4}
=			
z_{c1}	z_{c2}	z_{c3}	z_{c4}

... ...

- Full vectorization if no branches
- Scales with big number of tasks

AoS vs SoA

Array of Structures (AoS)

```
struct TDataElement {
  float x, y, z;
};

TDataElement data[8];
```

x y z x y z x y z x y z x y z x y z

- Vectorization needs data movement
- Padding wastes the memory
+ x, y, z in the same cache line
+ Intuitively structured code

Structure of Arrays (SoA)

```
struct TData {
  float x[8], y[8], z[8];
};

TData data;
```

x x x x x x . . y y y y y y . . z z z z z z . .

+ The data is already grouped for vectorisation
+ Compact data placement
- x, y, z in tree different cache lines
- Confusing code

Array of Structures of Arrays

```
struct TDataVecElement {
  float x[vecLen], y[vecLen], z[vecLen];
};

TDataVecElement data[2];
```

x x x x y y y y z z z z x x x x . . .

Matrix Problem

Exersices/2_SIMD/0_Matrix

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad c = \begin{bmatrix} \sqrt{a_{00}} & \sqrt{a_{01}} & \sqrt{a_{02}} \\ \sqrt{a_{10}} & \sqrt{a_{11}} & \sqrt{a_{12}} \\ \sqrt{a_{20}} & \sqrt{a_{21}} & \sqrt{a_{22}} \end{bmatrix} = ?$$

Scalar

```
for( int i = 0; i < N; i++ ) {
  for( int j = 0; j < N; j++ ) {
      c[i][j] = sqrt(a[i][j]);
  }
}
```

SIMD

?

Task:
- Write the SIMD part of the code using the SIMD-header file.
- Compare time and results.
- Can you avoid copying?

Quadratic Equations Problem

Exersices/2_SIMD/1_QuadraticEquation

$$ax^2 + bx + c = 0 \qquad x_{\max} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = ?$$

Scalar

```
for(int i=0; i<N; i++)  {
      float det = b[i]*b[i] - 4*a[i]*c[i];
      x[i] = (-b[i]+sqrt(det))/(2*a[i]);
}
```

SIMD

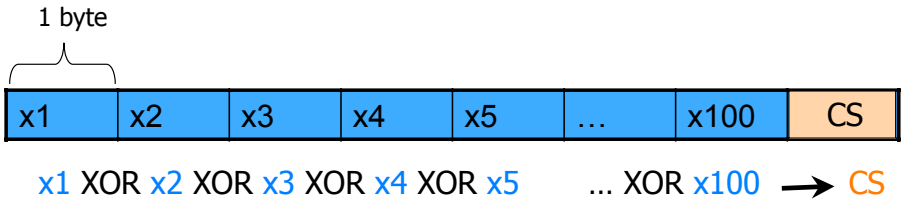
?

Task:

- Write the SIMD code for the root calculation using SIMD intrinsics and copying the data to SIMD vectors.
- Write the SIMD code using SIMD intrinsics and casting the data from the scalar arrays to SIMD vectors (use reinterpret_cast for this). Compare the time with a previous task.
- Write the SIMD code using header files and copying the data to SIMD vectors. Compare the time with previous tasks.
- Write the SIMD code using header files and casting the data from the scalar arrays to SIMD vectors (use reinterpret_cast for this). Compare the time with previous tasks.
- Put NVectors = 10000000; and NIterOut=10. Compare times and speed up factors with the previous results.

Check sum

Exersices/2_SIMD/2_CheckSum



x1 XOR x2 XOR x3 XOR x4 XOR x5 … XOR x100 → CS

```
template< typename T >
T Sum(const T* data, const int N)
{
   T sum = 0;
   const T* end = data + N;
   for ( int i = 0; i < N; ++i )
     sum = sum ^ data[i];
   return sum;
}
```

How to vectorize?
What is max speed up?
Is it possible to vectorize without SIMD-intrinsics?

2.1. SIMD with headers

Exercises are located at Exercises/2_SIMD/
Solutions are located at Exercises/2_SIMD/Solutions/
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

SIMD Introduction

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. SIMD instructions can be found, to one degree or another, on most CPUs, including the Intel's SSE, AVX, AMD's 3DNow!, IBM's AltiVec, SPE for PowerPC etc.

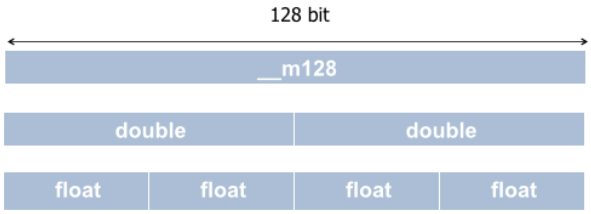


Fig. 1. Typical SIMD register.

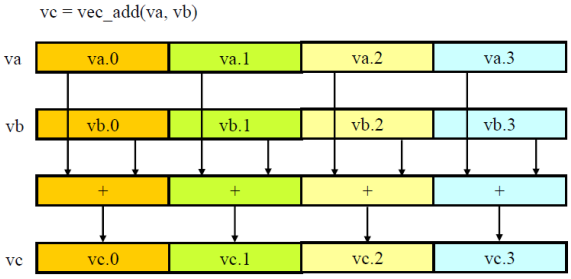


Fig. 2. The explanation of vec_add intrinsic.

To use C++ with SIMD support on a typical Intel CPU one needs to load floating point data into the special **__m128** data type, which contains 4 floating point variables. Then Streaming SIMD Extensions (SSE) intrinsics can be applied to the **__m128** data type as a usual functions. A list of the most common SSE instructions for single precision is given in Table 1.

For convenience the instructions can be wrapped by C++ class, which overloads standard operators (+, -, *, \, >, etc.). A small header file with

SSE	Scalar analog
_mm_set_ps(a3,a2,a1,a0)	convert 4 floats into SSE vector
_mm_add_ps (Va,Vb)	a + b
_mm_sub_ps (Va,Vb)	a - b
_mm_mul_ps (Va,Vb)	a * b
_mm_div_ps (Va,Vb)	a / b
_mm_and_ps (Va,Vb)	a & b
_mm_or_ps (Va,Vb)	a | b
_mm_xor_ps (Va,Vb)	a ^ b
_mm_cmplt_ps (Va,Vb)	a < b
_mm_cmple_ps (Va,Vb)	a <= b
_mm_cmpgt_ps (Va,Vb)	a > b
_mm_cmpge_ps (Va,Vb)	a >= b
_mm_cmpeq_ps (Va,Vb)	a == b
_mm_min_ps (Va,Vb)	min (a, b)
_mm_max_ps (Va,Vb)	max (a, b)
_mm_rcp_ps (Va)	1/a
_mm_sqrt_ps (Va)	sqrt(a)
_mm_rsqrt_ps (Va)	1/sqrt(a)

Table. 1. Common SSE instructions

overloaded instructions is provided at vectors/P4_F32vec4.h. It wraps **__m128** data type by **F32vec4** object, providing **operator[]** for access and operators like **operator+** and **operator<** for computations. Also a more general short name **fvec** is given to the **F32vec4** type, the vector length is saved to **fvecLen** and the scalar entry type to **fscal**. See part of the header file below.

Part of the source code of P4_F32vec4.h

```
44  class F32vec4
45  {
46   public:
47
48    __m128 v;
49
50    float & operator[]( int i ){ return (reinterpret_cast<float*>(&v))[i]; }
51    float   operator[]( int i ) const { return (reinterpret_cast<const
    float*>(&v))[i]; }
52
53    F32vec4( ):v(_mm_set_ps1(0)){}
54    F32vec4( const __m128 &a ):v(a) {}
55    F32vec4( const float &a ):v(_mm_set_ps1(a)) {}
56
57    F32vec4( const float &f0, const float &f1, const float &f2, const float
    &f3 ):v(_mm_set_ps(f3,f2,f1,f0)) {}
…
62    /* Arithmetic Operators */
63    friend F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return
    _mm_add_ps(a,b); }
…
72    /* Square Root */
73    friend F32vec4 sqrt ( const F32vec4 &a ){ return _mm_sqrt_ps (a); }
…
214 } __attribute__ ((aligned(16)));
215
216
217 typedef F32vec4 fvec;
218 typedef float  fscal;
219 const int fvecLen = 4;
```

Headers with vector instructions emulated by scalar operations are provided in **PSEUDO_F32vec4.h** (vector with 4 entires) and **PSEUDO_F32vec1.h** (vector with 1 entry). They can be used in the similar way as **P4_F32vec4.h**, but won't give any speed up. They supposed to be used for debugging and comparison.

2_SIMD/0_Matrix: description

The Matrix exercise requires to parallelize square root extraction over a set of float variables arranged in a square matrix, see Fig. 1. The initial code gives an implemented scalar part and lives blank space for a vector part. Therefore the initial output shows 0 time for vector calculations and infinite speed up factor, which should be currently ignored.

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad c = \begin{bmatrix} \sqrt{a_{00}} & \sqrt{a_{01}} & \sqrt{a_{02}} \\ \sqrt{a_{10}} & \sqrt{a_{11}} & \sqrt{a_{12}} \\ \sqrt{a_{20}} & \sqrt{a_{21}} & \sqrt{a_{22}} \end{bmatrix} = ?$$

Fig. 1. The explanation of Matrix task.

Part of the source code of Matrix.cpp

```
23  float a[N][N]; // input array
24  float c[N][N]; // output array for scalar computations
```

Part of the source code of Matrix.cpp

```
25  float c_simd[N][N]; // output array for SIMD computations
26
27  template<typename T> // required calculations
28  T f(T x) {
29    return sqrt(x);
30  }
…
45  int main() {
46
47      // fill classes by random numbers
48    for( int i = 0; i < N; i++ ) {
49      for( int j = 0; j < N; j++ ) {
50        a[i][j] = float(rand())/float(RAND_MAX); // put a random value, from 0 to
    1
51      }
52    }
53
54      /// -- CALCULATE --
55      /// SCALAR
56    TStopwatch timerScalar;
57    for( int ii = 0; ii < NIter; ii++ )
58      for( int i = 0; i < N; i++ ) {
59        for( int j = 0; j < N; j++ ) {
60          c[i][j] = f(a[i][j]);
61        }
62      }
63    timerScalar.Stop();
64
65      /// SIMD VECTORS
66    TStopwatch timerSIMD;
67      // TODO
68    timerSIMD.Stop();
69
70    double tScal = timerScalar.RealTime()*1000;
71    double tSIMD1 = timerSIMD.RealTime()*1000;
72
73    cout << "Time scalar: " << tScal << " ms " << endl;
74    cout << "Time SIMD:   " << tSIMD1 << " ms, speed up " << tScal/tSIMD1 <<
    endl;
75
76    CheckResults(c,c_simd);
77
78    return 1;
79  }
```

Typical output

```
Time scalar: 599.968 ms
Time SIMD:   0 ms, speed up inf
ERROR! SIMD and scalar results are not the same.
```

2_SIMD/0_Matrix: solution

For vector calculations all N*N scalar variables need to be divided into N*(N/4) groups with 4 elements, that will be treated as vectors. The solution groups scalars within each row, thus creating a matrix with N rows and N/4 columns.

Initial data copying into vectors can be avoided by reinterpreting them as vectors in case the scalar data is aligned to 16 bytes (line 23 of the solution). In the same way copying can be avoided at the output stage (line 25).

Although one can use the same algorithm structure with a triple loop as in the scalar code, we iterate over 4 elements groups rather than over each matrix element, therefore the step of the innermost loop need to be modified.

Part of the source code of Matrix_solution.cpp

```
23  float a[N][N] __attribute__((aligned(16)));      // input array
24  float c[N][N];                                   // output array for scalar
    computations
25  float c_simd[N][N] __attribute__((aligned(16))); // output array for SIMD
    computations
…
66    TStopwatch timerSIMD;
67    for( int ii = 0; ii < NIter; ii++ )
68      for( int i = 0; i < N; i++ ) {
69        for( int j = 0; j < N; j+=fvecLen ) {
70            fvec &aVec = reinterpret_cast<fvec&>(a[i][j]);
71            fvec &cVec = reinterpret_cast<fvec&>(c_simd[i][j]);
72            cVec = f(aVec);
73        }
74      }
75    timerSIMD.Stop();
```

Typical output after solution

```
Time scalar: 604.324 ms
Time SIMD:   150.4 ms, speed up 4.01811
SIMD and scalar results are the same.
```

Since the function **f(…)**, which must be applied to the data, is a template, we can use the same function call format.

Since 4 float variables fit into a single SIMD vector, all calculations are done in parallel and no overhead operations are required, the expected speed-up factor should be 4.

2_SIMD/1_QuadraticEquation: description

The QuadraticEquation exercise requires to vectorize solution of a set of quadratic equations in four different ways: using either (1) copying of data or (2) casting and using either (3) SIMD intrinsics or (4) **fvec** type from the header file. Then (5) compare the calculation time depending on amount of processed data.

It is recommended to compile the code with **-fno-tree-vectorize** option to prevent auto-vectorization of the scalar code, otherwise comparison of the vectorized and scalar codes will not be direct.

Part of the source code of QuadraticEquation.cpp

```
74    // fill parameters by random numbers
75    for( int i = 0; i < N; i++ ) {
76      a[i] = float(rand())/float(RAND_MAX); // put a random value, from 0 to 1
```

Part of the source code of QuadraticEquation.cpp

```
77      b[i] = float(rand())/float(RAND_MAX);
78      c[i] = -float(rand())/float(RAND_MAX);
79    }
80
81      /// -- CALCULATE --
82
83    // scalar calculations
84    TStopwatch timerScalar;
85    for(int io=0; io<NIterOut; io++)
86      for(int i=0; i<N; i++)
87      {
88        float det = b[i]*b[i] - 4*a[i]*c[i];
89        x[i] = (-b[i]+sqrt(det))/(2*a[i]);
90      }
91    timerScalar.Stop();
```

Typical output

```
Time scalar: 431.702 ms
Time SIMD1:   0 ms, speed up inf
Time SIMD2:   0.000953674 ms, speed up 452672
Time SIMD3:   0 ms, speed up inf
Time SIMD4:   0.000953674 ms, speed up 452672
ERROR! SIMD1 and scalar results are not the same.
ERROR! SIMD2 and scalar results are not the same.
ERROR! SIMD3 and scalar results are not the same.
ERROR! SIMD4 and scalar results are not the same.
```

2_SIMD/1_QuadraticEquation: solution

(1) The first task is to vectorize calculations using copying of data and SIMD intrinsics.

One need to use **__m128** type and **_mm_set_ps** function to copy the data. This function takes four float arguments and fills 4 32-bit elements in **__m128** variable starting from the last one. Therefore for i-th vector one uses (i*fvecLen+3)-th, (i*fvecLen+2)-th, (i*fvecLen+1)-th and (i*fvecLen)-th scalars as arguments.

The **_mm_set_ps1** function is used to convert constants 4 and 2 into the SIMD type. The **_mm_sub_ps**, **_mm_mul_ps**, **_mm_sqrt_ps** and **_mm_div_ps** are used for all required calculations.

Part of the source code of QuadraticEquation_solution.cpp

```
98        __m128 aV = _mm_set_ps(a[i*fvecLen+3],a[i*fvecLen+2],a[i*fvecLen
          +1],a[i*fvecLen]);
99        __m128 bV = _mm_set_ps(b[i*fvecLen+3],b[i*fvecLen+2],b[i*fvecLen
          +1],b[i*fvecLen]);
100       __m128 cV = _mm_set_ps(c[i*fvecLen+3],c[i*fvecLen+2],c[i*fvecLen
          +1],c[i*fvecLen]);
101
102       const __m128 det = _mm_sub_ps(_mm_mul_ps(bV,bV) ,
          _mm_mul_ps(_mm_set_ps1(4),_mm_mul_ps(aV,cV) ) );
103       __m128 xV =
          _mm_div_ps(_mm_sub_ps(_mm_sqrt_ps(det),bV),_mm_mul_ps(_mm_set_ps1(2),aV));
```

(2) The second task is vectorisation using cast and SIMD intrinsics.

Since the data is already aligned to 16 bytes one can use **reinterpret_cast** directly to the part of the scalar input and output arrays starting from the i-th element.

The calculation part remains the same.

Part of the source code of QuadraticEquation_solution.cpp

```
114     for(int i=0; i<N; i+=fvecLen)
115     {
116       __m128& aV = (reinterpret_cast<__m128&>(a[i]));
117       __m128& bV = (reinterpret_cast<__m128&>(b[i]));
118       __m128& cV = (reinterpret_cast<__m128&>(c[i]));
119
120       __m128& xV = (reinterpret_cast<__m128&>(x_simd2[i]));
```

(3) The third task is to vectorize calculations using copying of data and headers.

With headers one uses **fvec** type and **fvec** constructor to copy the data. The constructor takes 4 float arguments and fills 4 32-bit elements starting from the first one. Therefore for the i-th vector one uses (i*fvecLen)-th, (i*fvecLen+1)-th, (i*fvecLen+2)-th and (i*fvecLen+3)-th scalars as arguments.

The calculations part remains the same.

Part of the source code of QuadraticEquation_solution.cpp

```
134       fvec aV = fvec(a[i*fvecLen],a[i*fvecLen+1],a[i*fvecLen+2],a[i*fvecLen
      +3]);
135       fvec bV = fvec(b[i*fvecLen],b[i*fvecLen+1],b[i*fvecLen+2],b[i*fvecLen
      +3]);
136       fvec cV = fvec(c[i*fvecLen],c[i*fvecLen+1],c[i*fvecLen+2],c[i*fvecLen
      +3]);
137
138       const fvec det = bV*bV - 4*aV*cV;
139       fvec xV = (-bV+sqrt(det))/(2*aV);
```

(4) The fourth task is to vectorize using cast of data and headers. It can be done as in the tasks (2) and (3).

Typical output after solution

```
Time scalar: 432.194 ms
Time SIMD1:   109.914 ms, speed up 3.93211
Time SIMD2:   105.318 ms, speed up 4.1037
Time SIMD3:   111.117 ms, speed up 3.88954
Time SIMD4:   105.588 ms, speed up 4.09321
SIMD1 and scalar results are the same.
SIMD2 and scalar results are the same.
SIMD3 and scalar results are the same.
SIMD4 and scalar results are the same.
```

It is expected to have exactly the same speed up factor independently of SSE intrinsics overloading. The speed up factor of 4 should be achieved with **reinterpret_cast**. It should be slightly less in the other case, because additional time is spend on data management.

(5) The initial number of vectors is 10,000, which gives the size of arrays 10000*16 bytes = 0.16 MB. All 5 arrays should fit into the cache memory (typical size 1-20 MB), therefore the speed-up factor of 4 should be achievable with **reinterpret_cast**.

When the number of vectors is increased by a factor of 1000, the size of arrays becomes 160 MB, then the arrays will be stored in the much slower RAM memory. Therefore the speed-up factor will be less than 4. The speed-up factor should remain the same (low) also for 1,000,000 vectors.

Typical output after solution

```
Time scalar: 4322.71 ms
```

Typical output after solution

```
Time SIMD1:   1197.12 ms, speed up 3.61093
Time SIMD2:   1130.53 ms, speed up 3.82361
Time SIMD3:   1215.68 ms, speed up 3.5558
Time SIMD4:   1137.61 ms, speed up 3.79981
SIMD1 and scalar results are the same.
SIMD2 and scalar results are the same.
SIMD3 and scalar results are the same.
SIMD4 and scalar results are the same.
```

2_SIMD/2_CheckSum: description

The CheckSum exercise requires to vectorize the check sum calculation for the array of data. Check sum is defined as XOR-sum over all bytes of the array (lines 29-30 in the code below). In addition, consider if parallel calculations here can be done without the hardware support of parallelization.

Part of the source code of CheckSum.cpp

```
23 template< typename T >
24 T Sum(const T* data, const int N)
25 {
26     T sum = 0;
27 
28     for ( int i = 0; i < N; ++i )
29       sum = sum ^ data[i];
30     return sum;
31 }
32 
33 int main() {
34 
35     // fill string by random values
36   for( int i = 0; i < N; i++ ) {
37     str[i] = 256 * ( double(rand()) / RAND_MAX ); // put a random value, from 0
   to 255
38   }
39 
40     /// -- CALCULATE --
41 
42     /// SCALAR
43 
44   unsigned char sumS = 0;
45   TStopwatch timerScalar;
46   for( int ii = 0; ii < NIter; ii++ )
47     sumS = Sum<unsigned char>( str, N );
48   timerScalar.Stop();
```

2_SIMD/2_CheckSum: solution

The parallelization can be achieved, since XOR operator is applied bitwise and can be applied to the SIMD register data in the same way as to **char** variable, such that we can use **__m128** type to pack 16 **char** variables into a vector and treat them simultaneously.

The input **char** array should be reinterpreted as an array of **fvec**s. Then XOR sum of **fvec**s is calculated using the given template function. The resulting **fvec** variable should be reinterpreted back as an array of 16 **char** variables and the scalar sum of 16 elements is calculated directly.

Part of the source code of CheckSum_solution.cpp

```
52    unsigned char sumV = 0;
53
54    const int fvecCharLen = fvecLen*4;
55    const int NV = N/fvecCharLen;
56
57    TStopwatch timerSIMD;
58    for( int ii = 0; ii < NIter; ii++ ) {
59      fvec sumVV = 0;
60      sumVV = Sum<fvec>( reinterpret_cast<fvec*>(str), NV );
61      unsigned char *sumVS = reinterpret_cast<unsigned char*>(&sumVV);
62
63      sumV = sumVS[0];
64      for ( int iE = 1; iE < fvecCharLen; ++iE )
65        sumV ^= sumVS[iE];
66    }
67    timerSIMD.Stop();
```

Parallelization without SIMD instructions can be achieved, since XOR operator is applied bitwise and can be applied to integer data in the same way as to SIMD data and byte data. The only difference is the length of the vector, that is 4 in case of integer.

Part of the source code of CheckSum_solution.cpp

```
71    unsigned char sumI = 0;
72
73    const int intCharLen = 4;
74    const int NI = N/intCharLen;
75
76    TStopwatch timerINT;
77    for( int ii = 0; ii < NIter; ii++ ) {
78      int sumII = Sum<int>( reinterpret_cast<int*>(str), NI );
79      unsigned char *sumIS = reinterpret_cast<unsigned char*>(&sumII);
80
81      sumI = sumIS[0];
82      for ( int iE = 1; iE < intCharLen; ++iE )
83        sumI ^= sumIS[iE];
84    }
85    timerINT.Stop();
```

Typical output after solution

```
Time scalar: 274.435 ms
Time INT:   65.3081 ms, speed up 4.20216
Time SIMD:   25.3839 ms, speed up 10.8114
Results are the same.
```

A typical speed-up with integer should be 4, as expected, since 4 bytes are packed into one integer and time for the additional loop with 3 iterations is negligible. The speed-up factor using **fvec** theoretically should be 16, practically the achieved speed-up factor is about 10 to 12.

Typical output of KFLineFitter_solution2_simd.cpp

```
 Begin
Time: 0.647068 ms
 End
```

HPC Practical Course
Part 2.2

Kalman Filter Track Fit

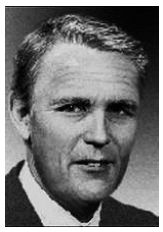
V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak

Goethe University of Frankfurt am Main

6 May 2014

The Kalman Filter

The Kalman filter is a recursive algorithm which estimates the state of a dynamic system from a series of incomplete and noisy measurements.



The filter is named after Rudolf E. Kalman.

The filter was developed in papers by Swerling (1958), Kalman (1960), and Kalman and Bucy (1961).

An example of an application would be to provide accurate continuously-updated information about the position and velocity of an object given only a sequence of observations about its position, each of which includes some error.

It is used in a wide range of engineering applications from radar to computer vision.

A wide variety of Kalman filters have now been developed, from Kalman's original formulation, now called the *simple* Kalman filter, to *extended* filter, the *information* filter and a variety of *square-root* filters.

The Kalman Filter Algorithm

The Kalman filter is a recursive estimator – only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state.

mean value over n measurements

n

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

previous estimation

new measurement

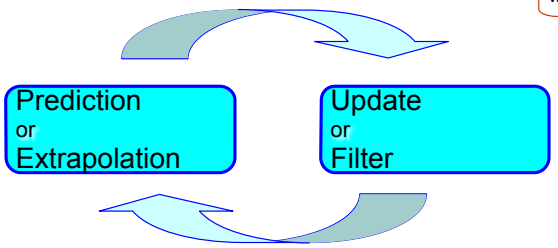
n+1

$$\mu_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{n}{n+1}\left(\frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n}x_{n+1}\right) = \frac{n}{n+1}\mu_n + \frac{1}{n+1}x_{n+1} = \mu_n + \frac{1}{n+1}(x_{n+1} - \mu_n)$$

mean value over n+1 measurements

weight

correction



The Kalman filter uses the dynamics of the target, which describes its time evolution, to remove the effects of the noise and get a good estimate of the location of the target

- at the present time (filtering),
- at a future time (prediction), or
- at a time in the past (interpolation or smoothing).

Components of the Kalman Filter method

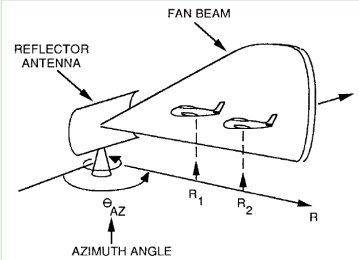
- **State vector:** contains a set of parameters to define system state in a certain time:

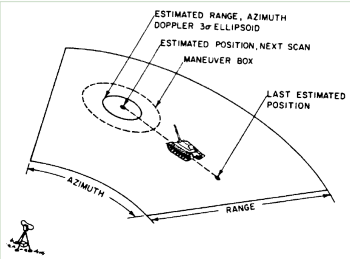
$$r = \{ x, y, z, v_x, v_y, v_z \}$$

- **Covariance matrix:** contains estimates of the uncertainty and correlation between uncertainties of state vector components. Based on information supplied to the Kalman filter. Not obtained from measurements. :

$$C = \begin{Bmatrix} \sigma^2_x & \sigma_{x\,y} & \sigma_{x\,z} & \sigma_{x\,vx} & \sigma_{x\,vy} & \sigma_{x\,vz} \\ \sigma_{x\,y} & \sigma^2_y & \sigma_{y\,z} & \sigma_{y\,vx} & \sigma_{y\,vy} & \sigma_{y\,vz} \\ \sigma_{x\,z} & \sigma_{y\,z} & \sigma^2_z & \sigma_{z\,vx} & \sigma_{z\,vy} & \sigma_{z\,vz} \\ \sigma_{x\,vx} & \sigma_{y\,vx} & \sigma_{z\,vx} & \sigma^2_{vx} & \sigma_{vx\,vy} & \sigma_{vx\,vz} \\ \sigma_{x\,vy} & \sigma_{y\,vy} & \sigma_{z\,vy} & \sigma_{vx\,vy} & \sigma^2_{vy} & \sigma_{vz\,vy} \\ \sigma_{x\,vz} & \sigma_{y\,vz} & \sigma_{z\,vz} & \sigma_{vx\,vz} & \sigma_{vz\,vy} & \sigma^2_{vz} \end{Bmatrix}$$

Example: Radar Applications





In a radar application, where one is interested in following a target, information about the location, speed, and acceleration of the target is measured at different moments in time with corruption by noise.

State vector

$r = \{ x, y, z, v_x, v_y, v_z \}$

position velocity

error of x

Covariance matrix

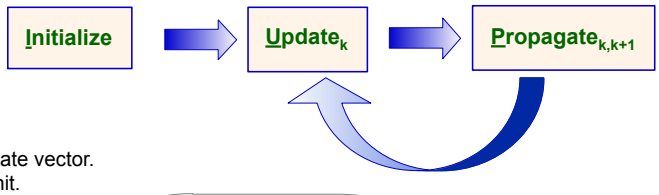
$$C = \begin{Bmatrix} \sigma^2_x & & & & & \\ & \sigma^2_y & & & \dots & \\ & & \sigma^2_z & & & \\ & & & \sigma^2_{vx} & & \\ & \dots & & & \sigma^2_{vy} & \\ & & & & & \sigma^2_{vz} \end{Bmatrix}$$

Apollo Flight Journal

December 21, 1968. The Apollo 8 spacecraft has just been sent on its way to the Moon.
003:46:31 Collins: Roger. At your convenience, would you please go P00 and Accept? We're going to update to your W-matrix.

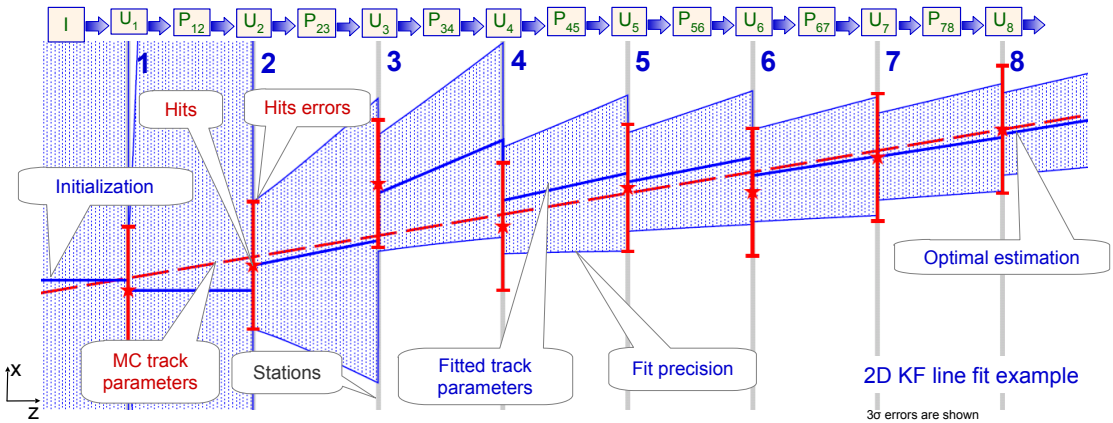
Kalman Filter Based Track Fit

Track fit: Optimal estimation of the track parameters according to hits – Kalman Filter



Kalman Filter:
1. Choose a state vector.
2. Start with an arbitrary initialization.
3. Add one hit after another, improving the state vector.
4. Get the optimal parameters after the last hit.

State vector of track parameters: $r = \{ x, t_x \}$ — position & direction



2D KF line fit example

3σ errors are shown

Kalman Filter Equation

$$\mathbf{r} = \mathbf{r}^t + \xi,$$
$$C = <\xi \cdot \xi^T>$$

How does measurement depend on the state vector?

Model of measurement

$$\mathbf{m}_k = H_k \mathbf{r}_k^t + \eta_k$$

How to propagate the state vector?

Prediction matrix (Jacobian)

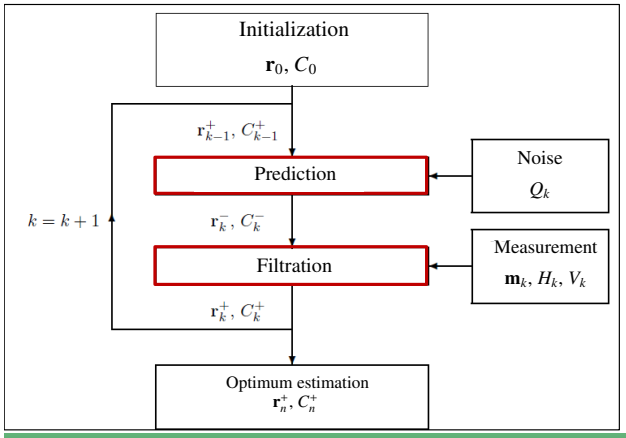
Noise

$$\mathbf{r}_k^t = F_{k-1}\mathbf{r}_{k-1}^t + \mathbf{v}_k$$

Prediction

$$\mathbf{r}_k^- = F_{k-1}\mathbf{r}_{k-1}^+$$
$$C_k^- = F_{k-1}C_{k-1}^+F_{k-1}^T + Q_k$$

Noise matrix



Filtering

Measurement error

Weighting matrix

$$S_k = (V_k + H_k C_k^- H_k^T)^{-1}$$

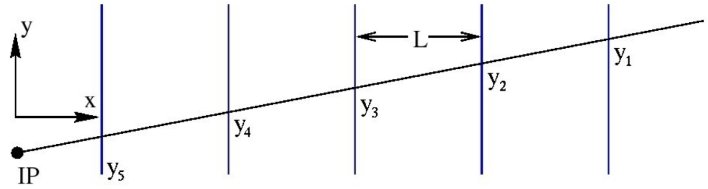
Gain matrix

$$K_k = C_k^- H_k^T S_k$$

Residual

$$\zeta_k = \mathbf{m}_k - H_k \mathbf{r}_k^-$$
$$\mathbf{r}_k^+ = \mathbf{r}_k^- + K_k \zeta_k$$
$$C_k^+ = (I - K_k H_k) \cdot C_k^-$$ Covariance matrix
$$\chi_k^2 = \chi_{k-1}^2 + \zeta_k^T S_k \zeta_k$$

Application of Kalman Filter to the Straight Line 2D Fit



The state vector and its covariance matrix:

$$\mathbf{r} = \begin{pmatrix} y \\ t_y \end{pmatrix}, \; C = \begin{pmatrix} C_{yy} & C_{yt_y} \\ C_{yt_y} & C_{t_y t_y} \end{pmatrix}$$

The only measurement and its covariance:

$$\mathbf{m}_k = \{y_k^m\}, \; V_k = \sigma^2$$

Model of the measurement:

$$H_k = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

$y_k = H_k r_k$

Prediction matrix:

$$F_{k-1} = \begin{pmatrix} \frac{\partial y_k^-}{\partial y_{k-1}^+} & \frac{\partial y_k^-}{\partial t_{y_{k-1}}^+} \\ \frac{\partial t_{y_k}^-}{\partial y_{k-1}^+} & \frac{\partial t_{y_k}^-}{\partial t_{y_{k-1}}^+} \end{pmatrix} = \begin{pmatrix} 1 & -L \\ 0 & 1 \end{pmatrix}$$

$y_k = y_{k-1} - L\, t_{yk}$

$t_{yk} = t_{yk-1}$

Application of Kalman Filter to the Line Fit: continue

Prediction (propagation):

$$r_k^- = \begin{pmatrix} y_{k-1}^+ - t_{y_{k-1}}^+ L, \\ t_{y_{k-1}}^+ \end{pmatrix},$$

$$C_k^- = \begin{pmatrix} C_{k-1_{yy}}^+ - 2LC_{k-1_{yt_y}}^+ + L^2 C_{k_{t_yt_y}}^+ & C_{k-1_{yt_y}}^+ - LC_{k-1_{t_yt_y}}^+ \\ C_{k-1_{yt_y}}^+ - LC_{k-1_{t_yt_y}}^+ & C_{k-1_{t_yt_y}}^+ \end{pmatrix}$$

Filtration:

$$S_k = \frac{1}{\sigma^2 + C_{k_{yy}}^-},$$

$$K_k = \frac{1}{\sigma^2 + C_{k_{yy}}^-} \begin{pmatrix} C_{k_{yy}}^- \\ C_{k_{yt_y}}^- \end{pmatrix},$$

$$\zeta_k = y_k^m - y_k^-,$$

$$\mathbf{r}_k^+ = \begin{pmatrix} y_k^- + \frac{C_{k_{yy}}^-}{\sigma^2 + C_{k_{yy}}^-}(y_k^m - y_k^-) \\ t_{y_k}^- + \frac{C_{k_{yt_y}}^-}{\sigma^2 + C_{k_{yy}}^-}(y_k^m - y_k^-) \end{pmatrix},$$

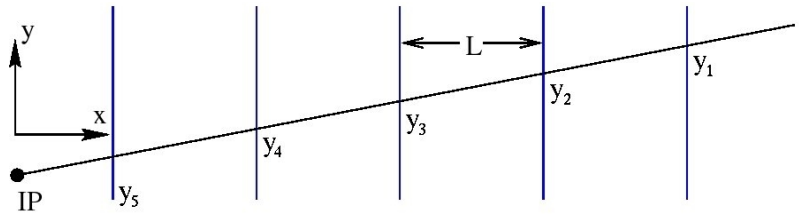
$$C_k^+ = \begin{pmatrix} C_{k_{yy}}^-(1 - \frac{C_{k_{yy}}^-}{\sigma^2 + C_{k_{yy}}^-}) & C_{k_{yt_y}}^-(1 - \frac{C_{k_{yy}}^-}{\sigma^2 + C_{k_{bb}}^-}) \\ C_{k_{yt_y}}^-(1 - \frac{C_{k_{yy}}^-}{\sigma^2 + C_{k_{yy}}^-}) & C_{k_{t_yt_y}}^- - \frac{C_{k_{yt_y}}^{-2}}{\sigma^2 + C_{k_{yy}}^-} \end{pmatrix}.$$

Pricise y-measurement:
σ = 0 : $y_k^+ = y_k^m$
$C_k^+{}_{yy} = 0$

Very inprecise measurement:
σ = inf : $r_k^+ = r_k^-$
$C_k^+ = C_k^-$

Exercise

Exersices/2_SIMD/3_KF



Compile and run:

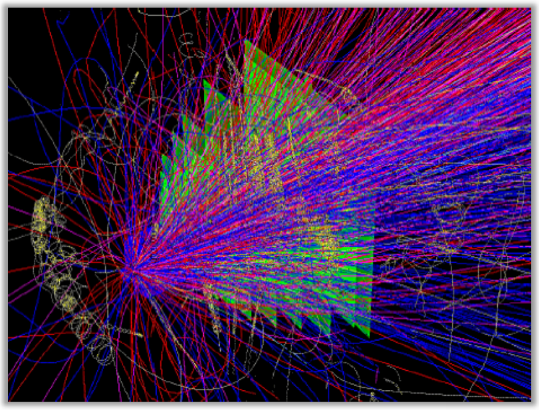
g++ KFLineFitter.cpp -O3; ./a.out

Check results

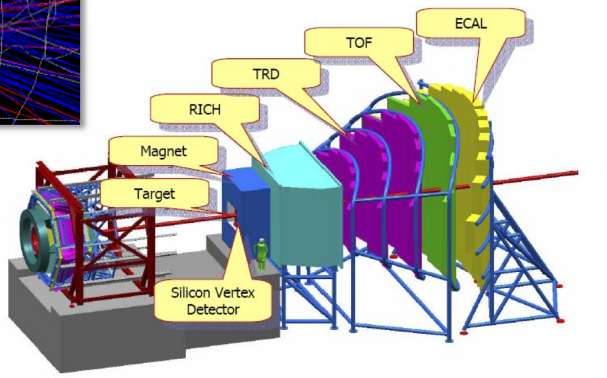
output - output file generated by KFfit programm.

Task:
SIMDize track fitting procedure using templates.
Compare results and time.

Future Experiment: CBM (FAIR/GSI)



- Fixed-target heavy-ion experiment
- 10^7 Au+Au collisions/s
- 1000 charged particles/collision
- Non-homogeneous magnetic field



No simple trigger primitive, like high p_t, available to tag events of interest. The only selective signature is the detection of the decay vertex.

Reconstruction packages:

- Track finding — Cellular Automaton (CA)
- Track fitting — Kalman Filter (KF)
- Vertexing — KFParticle

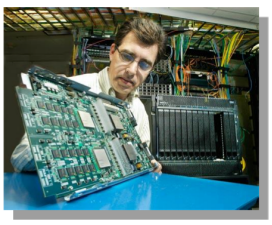
Full event reconstruction is required at the trigger level

Kalman Filter Track Fit on Cell

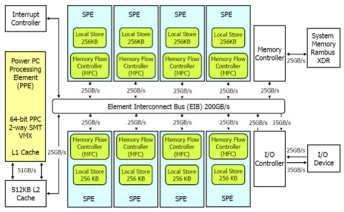
	Stage	Description	Time/track	Speedup
		Initial scalar version	12 ms	–
Intel P4	1	Approximation of the magnetic field	240 μs	50
Intel P4	2	Optimization of the algorithm	7.2 μs	35
Intel P4	3	Vectorization	1.6 μs	4.5
Cell	4	Porting to SPE	1.1 μs	1.5
Cell	5	Parallelization on 16 SPEs	0.1 μs	10
		Final simdized version	0.1 μs	120000

10000x faster on each CPU

Comp. Phys. Comm. 178 (2008) 374-383



The KF speed was increased by 5 orders of magnitude



blade11bc4 @IBM, Böblingen: 2 Cell Broadband Engines with 256 kB Local Store at 2.4 GHz

Motivated by, but not restricted to Cell !

2.2. Kalman Filter Track Fit

Exercises are located at Exercises/2_SIMD/
Solutions are located at Exercises/2_SIMD/Solutions/
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

2_SIMD/3_KF: description

The Kalman filter is a method of obtaining estimate of unknown variable that uses a series of noisy measurements observed over time. The resulting estimate tend to be more precise than estimates based on a single measurements alone. The filter is named after Rudolf Kalman, one of the primary developers of its theory.

The Kalman filter has numerous applications in technology and science. A common application is for guidance, navigation and control of vehicles, particularly aircrafts and spacecrafts. Furthermore, the Kalman filter is a widely applied concept used in fields such as signal processing and econometrics.

The Kalman filter is a recursive estimator – only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state. For illustration let us reformulate a calculation of a mean value of N elements in a recursive form. The general form of mean value is defined as:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The mean over n+1 elements can be rewritten in a way:

$$\mu_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{n}{n+1}\left(\frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n}x_{n+1}\right) = \frac{n}{n+1}\mu_n + \frac{1}{n+1}x_{n+1} = \mu_n + \frac{1}{n+1}(x_{n+1} - \mu_n)$$

Now the mean of n+1 is determined by the previous estimate and a correction term, which is a new measurement with a weighting coefficient 1/(n+1). After we have reformulated the problem in a recursive way Kalman filter method becomes applicable.

The Kalman filter method is intended for finding the optimum estimation **r** of an unknown state vector of a system $\mathbf{r}^t$ based on k measurements m_k, k = 1,…,n by minimising the mean square estimation error. The estimation **r** is known with the error **ξ**:

$\mathbf{r} = \mathbf{r}^t + \boldsymbol{\xi}$,

therefore the covariance matrix of the estimation is introduced:

$C = \langle \boldsymbol{\xi} \cdot \boldsymbol{\xi}^T \rangle$.

The state vector is normally not observed directly, but through the detector measurements. Let's assume that the measurement $\mathbf{m}_k$ linearly depends on $\mathbf{r}^t_k$:

$\mathbf{m}_k = H_k \mathbf{r}^t_k + \boldsymbol{\eta}_k$, where H_k is the measurement model and $\boldsymbol{\eta}_k$ is an error of the k-th measurement.

The evolution of the linear system proceeds in space from one measurement $\mathbf{m}_{k-1}$ to the next measurement $\mathbf{m}_k$ and is described by a linear equation:

$\mathbf{r}^t_k = F_{k-1}\mathbf{r}^t_{k-1} + \mathbf{v}_k$,

where F_{k-1} is a linear propagation operator, $\mathbf{v}_k$ is a random process noise between the measurements $\mathbf{m}_{k-1}$ and $\mathbf{m}_k$.

The measurement errors $\boldsymbol{\eta}_k$ and the process noise $\mathbf{v}_k$ are assumed to be uncorrelated and unbiased, and those covariance matrices V_k and Q_k are known:

$\langle \boldsymbol{\eta} \rangle = \langle \mathbf{v} \rangle = 0$,

$\langle \boldsymbol{\eta}_k \cdot \boldsymbol{\eta}^T{}_k \rangle \equiv V_k$, (4b) $\langle \mathrm{v}_k \cdot \mathrm{v}^T{}_k \rangle \equiv Q_k$.

The conventional Kalman filter algorithm (details in Fig. 3) consists of three stages:

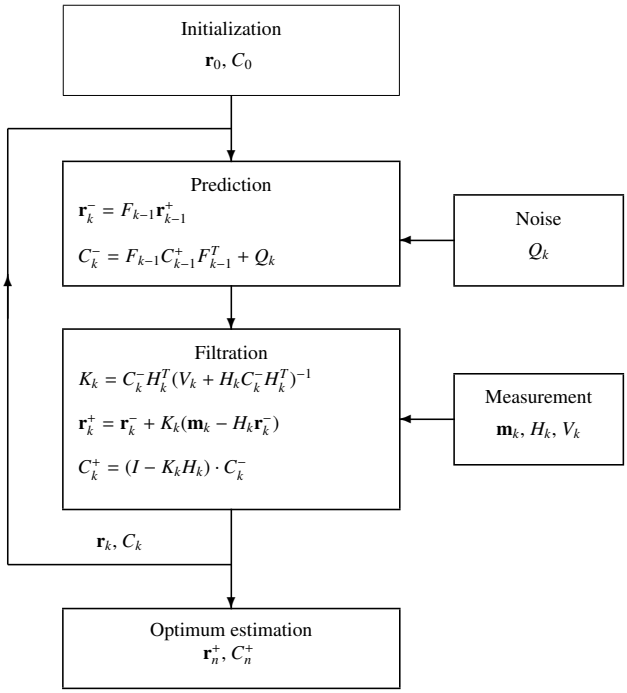


Fig. 3. Block diagram of the conventional Kalman filter.

Initialisation: The state vector $\mathbf{r}$ is initialised either arbitrary or with some approximate values. The covariance matrix is set to $C_0 = I \cdot \mathrm{inf}^2$, where inf denotes a large number.

Prediction: The current estimations of the state vector and the covariance matrix at the measurement $\mathbf{m}_{k-1}$ are propagated to the next measurement, and the process noise is taken into account. For the first propagation the initialisation values are used instead of a non-existed measurement.

Filtration: The predicted state vector and the covariance matrix are updated with the new measurement to get their optimal estimations, also at this stage we calculate $\boldsymbol{\zeta}_k$ – the residual, distance between the predicted and the actual measurement and W_k – the weight matrix, inverse covariance matrix of the residual:

$\zeta_k = \mathbf{m}_k - H_k \mathbf{r}_k$,

$W_k = (V_k + H_k C_k^- H_k^T)^{-1}$.

The following designations have been used: $\mathbf{r}^+{}_{k-1}$, C^+ – the optimum estimation and the error covariance matrix, obtained at the previous measurement; the matrix F_{k-1} relates the state at step $k-1$ to the state at step k; $\mathbf{r}^-{}_k$, C_k^- – predicted estimation of r_t and covariance matrix after the process noise; $\mathbf{m}_k$, V_k – the k-th measurement and its covariance matrix; the matrix H_k – the model of measurement; the value $\chi^2{}_k$ is the total χ^2-deviation of the obtained estimation $\mathbf{r}^+{}_k$ from the measurements $\mathbf{m}_1,\ldots,\mathbf{m}_k$.

The vector $\mathbf{r}^+{}_n$ obtained after the filtration of the last measurement is the desired optimal estimation of the $\mathbf{r}^t{}_n$ with the covariance matrix $C_n{}^+$.

In the our exercise we will deal with simple example of straight line trajectory in 2D space. Fig. 4 shows five detector planes placed along x axis. The distance between neighbouring detectors is L. We measure y-coordinate of a track in each detector plane with some error σ: (y_1, y_2, y_3, y_4, y_5). The task is to estimate the trajectory in the area of track origin IP. We will start estimation procedure with the last station 5. In this case of straight track the equation of motion is:

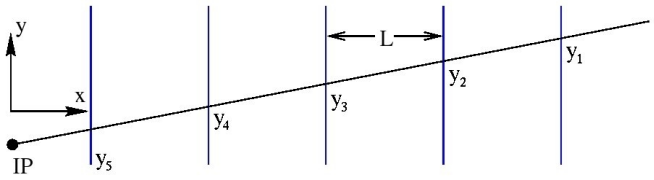


Fig. 4. Straight line track crossing detector planes.

$$y = t_y x + b.$$

Let us define state vector as the y-coordinate and tangent of track slope in x direction t_y and covariance matrix for this state vector:

$$\mathbf{r} = \begin{pmatrix} y \\ t_y \end{pmatrix},\ C = \begin{pmatrix} C_{yy} & C_{yt_y} \\ C_{yt_y} & C_{t_y t_y} \end{pmatrix}.$$

Since we measure only y-coordinate the measurement vector for this case and its' covariance matrix are:

$$\mathbf{m}_k = \{y_k^m\},\ V_k = \sigma^2.$$

The model of measurement for this case: $H_k = \begin{pmatrix} 1 & 0 \end{pmatrix}$.

So that: $y_k = H_k r_k$.

The propagation operator will be:

$$F_{k-1} = \begin{pmatrix} \frac{\partial y_k^-}{\partial y_{k-1}^+} & \frac{\partial y_k^-}{\partial t_{y_{k-1}}^+} \\ \frac{\partial t_{y_k}^-}{\partial y_{k-1}^+} & \frac{\partial t_{y_k}^-}{\partial t_{y_{k-1}}^+} \end{pmatrix} = \begin{pmatrix} 1 & -L \\ 0 & 1 \end{pmatrix}$$

So that the prediction stage can be rewritten in a way:

$y_k = y_{k-1} - L\, t_{yk}$.

$t_{yk} = t_{yk-1}$.

For the next vectorisation exercise we will have a closer look at KF algorithm in the case of straight line trajectory. The task is to SIMDise the estimation of particle tracks parameters using Kalman filter method[1]. The program consists of two parts: 1. simulation of particle tracks and 2. reconstruction of particle tracks parameters. An independent classes **LFSimulator** and **LFFitter** are present respectively for each task, both classes contain the parameters information of track environment, procedures to change those parameters and procedures to execute the task.

Only the reconstruction part must be vectorized. It is proposed to use templates for convenience of vectorisation and debugging, i.e. to create template classes and functions, which can be applied both to scalar and simd variables.

LFSimulator basing on given parameters of particle trajectory simulates interaction points with detector planes (Monte Carlo (MC) points) and detector measurements obtained due to these interactions (hits), they both structured into **LFTrack** class.

---

[1] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth and W.F.J. Müller, Fast SIMDized Kalman filter based track fit. CBM-SOFT-note-2007-001, 22 January 2007.

Part of the source code of KFLineFitter.cpp

```
41  struct LFPoint {
42    LFPoint():x(NAN0),z(NAN0){};
43    LFPoint( float x_, float z_ ): x(x_),z(z_) {};
44
45    float x; // x-position of the hit
46    float z; // coordinate of station
47  };
…
79  struct LFTrack {
80    vector<LFPoint> hits;
81
82    LFTrackParam rParam; // track parameters reconstructed by the fitter
83    LFTrackCovMatrix rCovMatrix; // error (or covariance) matrix
84    float chi2; // chi-squared deviation between points and trajectory
85    int ndf;    // number degrees of freedom
86
87    vector<LFTrackParam> mcPoints; // simulated track parameters
88  };
```

LFFitter basing on hits reconstructs parameters of particle trajectory and their error matrices (covariance matrices), chi-squared deviation between points and trajectory and number of degrees of freedom (NDF), which are also kept in **LFTrack** class.

Part of the source code of KFLineFitter.cpp

```
49  struct LFTrackParam {
…
59    float &X() { return p[0]; };
60    float &Tx() { return p[1]; };
61    float &Z() { return z; };
62
63    float p[2]; // x, tx.
64    float z;
…
67  };
…
177 void LFFitter::Fit( LFTrack& track ) const
178 {
179   Initialize( track );
180
181   const int NHits = track.hits.size();
182   for ( int i = 0; i < NHits; ++i ) {
183     Extrapolate( track, track.hits[i].z );
184     Filter( track, track.hits[i].x );
185   }
186
187   Extrapolate( track, track.mcPoints.back().z ); // exptrapolate to MC point
    for comparison with MC info
188 }
189
```

Part of the source code of KFLineFitter.cpp

```
190 void LFFitter::Initialize( LFTrack& track ) const
191 {
192   track.rParam.Z() = 0;
193   track.rParam.X() = 0;
194   track.rParam.Tx() = 0;
195   track.chi2 = 0;
196   track.ndf = -2;
197
198   track.rCovMatrix.C00() = InfX;
199   track.rCovMatrix.C10() = 0;
200   track.rCovMatrix.C11() = InfTx;
201 }
202
203 void LFFitter::Extrapolate( LFTrack& track, float z_ ) const
204 {
205   float &z = track.rParam.Z();
206   float &x = track.rParam.X();
207   float &tx = track.rParam.Tx();
208   float &C00 = track.rCovMatrix.C00();
209   float &C10 = track.rCovMatrix.C10();
210   float &C11 = track.rCovMatrix.C11();
211
212   const float dz = z_ - z;
213
214   x += dz * tx;
215   z = z_;
216
217   // F = 1  dz
218   //     0  1
219
220   const float C10_in = C10;
221   C10 += dz * C11;
222   C00 += dz * ( C10 + C10_in );
223 }
224
225 void LFFitter::Filter( LFTrack& track, float x_ ) const
226 {
227
228   float &x = track.rParam.X();
229   float &tx = track.rParam.Tx();
230   float &C00 = track.rCovMatrix.C00();
231   float &C10 = track.rCovMatrix.C10();
232   float &C11 = track.rCovMatrix.C11();
233
234     // H = { 1, 0 }
235     // zeta = Hr - r // P.S. not "r - Hr" here becase later will be rather "r =
    r - K * zeta" then  "r = r + K * zeta"
236   float zeta = x - x_;
237
238     // F = C*H'
```

Part of the source code of KFLineFitter.cpp

```
239   float F0 = C00;
240   float F1 = C10;
241
242     // H*C*H'
243   float HCH = F0;
244
245     // S = 1. * ( V + H*C*H' )^-1
246   float wi = 1./( fSigma*fSigma + HCH );
247   float zetawi = zeta * wi;
248
249   track.chi2 += zeta * zetawi ;
250   track.ndf  += 1;
251
252     // K = C*H'*S = F*S
253   float K0 = F0*wi;
254   float K1 = F1*wi;
255
256     // r = r - K * zeta
257   x  -= K0*zeta;
258   tx -= K1*zeta;
259
260     // C = C - K*H*C = C - K*F
261   C00 -= K0*F0;
262   C10 -= K1*F0;
263   C11 -= K1*F1;
264
265 }
```

2_SIMD/3_KF: solution

First of all it has to be decided which data should be grouped and how it should be grouped to vectorize the track fitting procedure. The grouped data should be maximally independent, therefore the most simple and effective way is to treat M (4) independent tracks in parallel. The procedure (see lines 177-188 in scalar version) would be the following: M tracks are initialised, M tracks are extrapolated to the 1-st station, M hits are taken into account in the tracks parameters estimation (one hit per track), M tracks extrapolated to 2-nd station, … M tracks extrapolated to z-coordinate of last mc point, which must be the same for all tracks.

To perform these procedures we should prepare hits grouping them from different tracks into one vector, all hits in group must be on the same station. Corresponding class should have vector of M x-coordinates of M hits and scalar of z-coordinate of M hits, which is same as z-coordinate of a station the hits belong to. The general type for M **float**s grouped together is noted as **T**. Both **fvec** and **float** types can be substituted here instead of **T**, that justifies the template construct usage.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
49  template< typename T >
50  struct LFPoint {
51    LFPoint():x(NAN0),z(NAN0){};
52    LFPoint( T x_, T z_ ): x(x_),z(z_) {};
53
54    T x; // x-position of the hit
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
55    float z; // coordinate of station // all points on one station have same z-
      position
56  };
```

Result of the procedure would be M track parameters grouped into one vectorized parameters class. Similarly to hits, **x** and **Tx** parameters and covariance elements are grouped together and z-coordinate is stays scalar.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
58  template< typename T >
59  struct LFTrackParam {
…
69    T &X() { return p[0]; };
70    T &Tx() { return p[1]; };
71    float &Z() { return z; };
72
73    T p[2]; // x, tx.
74    float z;
…
77  };
78
79  template< typename T >
80  struct LFTrackCovMatrix {
81    T &C00() { return c[0]; };
82    T &C10() { return c[1]; };
83    T &C11() { return c[2]; };
84
85    T c[3]; // C00, C10, C11
…
88  };
```

The data is grouped in track class, which also have additional chi-squared deviation and NDF, which can be different for different tracks, therefore required a vector type. Meanwhile NDF is integer, therefore additional parameter of template **I** is added for grouped integers.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
90  template< typename T, typename I >
91  struct LFTrack {
92    vector< LFPoint<T> > hits;
93
94    LFTrackParam<T> rParam; // reconstructed by the fitter track parameters
95    LFTrackCovMatrix<T> rCovMatrix; // error (or covariance) matrix
96    T chi2;   // chi-squared deviation between points and trajectory
97    I ndf;    // number degrees of freedom
98
99    vector< LFTrackParam<T> > mcPoints; // simulated track parameters
100 };
```

The same operations must be done in LFFitter functions, which implement data processing: basically all floats, with exception of z-coordinate, should be changed to template **T** type and all integers to **I** type, and

class types to the templates prepared for vector processing. Since 4 tracks are independent and similar no changes in the algorithm itself are required and the code is basically the same.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
231 template< typename T, typename I >
232 void LFFitter<T,I>::Fit( LFTrack<T,I>& track ) const
233 {
234   Initialize( track );
235   const int NHits = track.hits.size();
236   for ( int i = 0; i < NHits; ++i ) {
237     Extrapolate( track, track.hits[i].z );
238     Filter( track, track.hits[i].x );
239   }
240 
241   Extrapolate( track, track.mcPoints.back().z ); // just for pulls
242 }
243 
244 template< typename T, typename I >
245 void LFFitter<T,I>::Initialize( LFTrack<T,I>& track ) const
246 {
247   track.rParam.Z() = 0;
248   track.rParam.X() = 0;
249   track.rParam.Tx() = 0;
250   track.chi2 = 0;
251   track.ndf = -2;
252 
253   track.rCovMatrix.C00() = InfX;
254   track.rCovMatrix.C10() = 0;
255   track.rCovMatrix.C11() = InfTx;
256 }
257 
258 template< typename T, typename I >
259 void LFFitter<T,I>::Extrapolate( LFTrack<T,I>& track, float z_ ) const
260 {
261   float &z = track.rParam.Z();
262   T &x = track.rParam.X();
263   T &tx = track.rParam.Tx();
264   T &C00 = track.rCovMatrix.C00();
265   T &C10 = track.rCovMatrix.C10();
266   T &C11 = track.rCovMatrix.C11();
267 
268   const float dz = z_ - z;
269 
270   x += dz * tx;
271   z = z_;
272 
273   // F = 1  dz
274   //     0  1
275 
276   const T C10_in = C10;
277   C10 += dz * C11;
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
278   C00 += dz * ( C10 + C10_in );
279 }
280
281 template< typename T, typename I >
282 void LFFitter<T,I>::Filter( LFTrack<T,I>& track, T x_ ) const
283 {
284
285   T &x = track.rParam.X();
286   T &tx = track.rParam.Tx();
287   T &C00 = track.rCovMatrix.C00();
288   T &C10 = track.rCovMatrix.C10();
289   T &C11 = track.rCovMatrix.C11();
290
291     // H = { 1, 0 }
292     // zeta = Hr - r // P.S. not "r - Hr" here becase later will be rather "r =
    r - K * zeta" then  "r = r + K * zeta"
293   T zeta = x - x_;
294
295   // F = C*H'
296   T F0 = C00;
297   T F1 = C10;
298
299     // H*C*H'
300   T HCH = F0;
301
302     // S = 1. * ( V + H*C*H' )^-1
303   T wi = 1./( fSigma*fSigma + HCH );
304   T zetawi = zeta * wi;
305
306   track.chi2 += zeta * zetawi ;
307   track.ndf  += 1;
308
309     // K = C*H'*S = F*S
310   T K0 = F0*wi;
311   T K1 = F1*wi;
312
313     // r = r - K * zeta
314   x  -= K0*zeta;
315   tx -= K1*zeta;
316
317     // C = C - K*H*C = C - K*F
318   C00 -= K0*F0;
319   C10 -= K1*F0;
320   C11 -= K1*F1;
321
322 }
```

All the template classes and functions can be used for scalar calculations in the same way as before, just adding **<float,int>** template parameters to fit class:

Part of the source code of KFLineFitter_solution2_simd.cpp

```
361 #ifndef SIMDIZED
362
363   LFFitter<float,int> fit;
364
365   fit.SetSigma( Sigma );
366
367 #ifdef TIME
368   timer.Start(1);
369 #endif
370   for ( int i = 0; i < NTracks; ++i ) {
371     LFTrack<float,int> &track = tracks[i];
372     fit.Fit( track );
373   }
374   #ifdef TIME
375   timer.Stop();
376   #endif
377
378 #else
```

The **LFFitter** class can be used similarly for vectored computations, **T** parameter should be set to **fvec**, **I** parameter should be set to **fvec** either, since we can use floating point values to store integers. In addition the input data should be prepared and the output data should be converted to scalar format for future comparison. For this purpose one should introduce two additional functions: **CopyTrackHits** and **CopyTrackParams**.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
378 #else
379
380     // Convert scalar Tracks to SIMD-tracks
381   const int NVTracks = NTracks/fvecLen;
382   LFTrack<fvec,fvec> vTracks[NVTracks];
383
384   CopyTrackHits( tracks, vTracks, NVTracks );
385
386     // fit
387   LFFitter<fvec,fvec> fit;
388
389   fit.SetSigma( Sigma );
390
391 #ifdef TIME
392   timer.Start(1);
393 #endif
394   for ( int i = 0; i < NVTracks; ++i ) {
395     LFTrack<fvec,fvec> &track = vTracks[i];
396     fit.Fit( track );
397   }
398 #ifdef TIME
399   timer.Stop();
400 #endif
401
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
402     // Convert  SIMD-tracks to scalar Tracks
403   CopyTrackParams( vTracks, tracks, NVTracks );
404
405 #endif // SIMDIZED
```

The **CopyTrackHits** function is needed to copy all required by **LFFitter** class data into vectorized classes. These are full hits data and z-coordinate of the last MC point. To copy it one would need a loop over groups of tracks, **fvecLen** tracks in group (see line 384). For each group loop over track in group are required and a loop over hits in track (lines 111 and 114). Since all tracks have same number of hits, equal to number of stations we can take this number from the very first track and make it constant. The z-coordinate of the last point should be copied for each track after loop over hits.

The **CopyTrackParams** function is needed to copy all output data from vectorized classes to scalar classes. This would require similarly the loop over vectorized tracks, and loop over entries in the vectorized tracks, loop over parameters and loop over covariance matrix elements.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
103 void CopyTrackHits( const LFTrack<float,int>* sTracks, LFTrack<fvec,fvec>*
    vTracks, int nVTracks ){
104   const int NHits = sTracks[0].hits.size(); // all tracks have the same number
    of hits
105
106
107   for( int iV = 0; iV < nVTracks; ++iV ) {
108     LFTrack<fvec,fvec>& vTrack = vTracks[iV];
109     vTrack.hits.resize(NHits);
110     vTrack.mcPoints.resize(NHits);
111     for( int i = 0; i < fvecLen; ++i ) {
112       const LFTrack<float,int>& sTrack = sTracks[ iV*fvecLen + i ];
113
114       for( int iH = 0; iH < NHits; ++iH ) {
115         vTrack.hits[iH].x[i] = sTrack.hits[iH].x;
116         vTrack.hits[iH].z = sTrack.hits[iH].z;
117       }
118
119       vTrack.mcPoints[NHits-1].z = sTrack.hits[NHits-1].z; // need this info
    for comparison of reco and MC
120     }
121   }
122 }
123
124 void CopyTrackParams( const LFTrack<fvec,fvec>* vTracks, LFTrack<float,int>*
    sTracks, int nVTracks ){
125
126
127   for( int iV = 0; iV < nVTracks; ++iV ) {
128     const LFTrack<fvec,fvec>& vTrack = vTracks[iV];
129     for( int i = 0; i < fvecLen; ++i ) {
130       LFTrack<float,int>& sTrack = sTracks[ iV*fvecLen + i ];
131
132       for( int ip = 0; ip < 2; ++ip )
133         sTrack.rParam.p[ip] = vTrack.rParam.p[ip][i];
134       sTrack.rParam.z = vTrack.rParam.z;
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
135       for( int ic = 0; ic < 3; ++ic )
136         sTrack.rCovMatrix.c[ic] = vTrack.rCovMatrix.c[ic][i];
137       sTrack.chi2 = vTrack.chi2[i];
138       sTrack.ndf = vTrack.ndf[i];
139     }
140   }
141 }
```

The finial output, saved in the file must be the same for scalar and vector version. The time should be about factor 4 different.

Typical output of KFLineFitter.cpp

```
 Begin
Time: 2.35605 ms
 End
```

Typical output of KFLineFitter_solution2_simd.cpp

```
 Begin
Time: 0.647068 ms
 End
```

HPC Practical Course
Part 2.3

Vector classes (Vc)

V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak

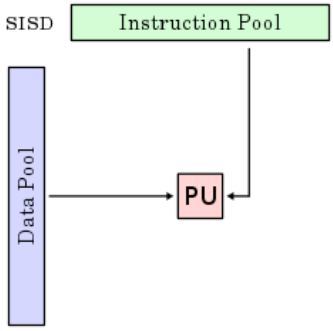
Goethe University of Frankfurt am Main

14 May 2014

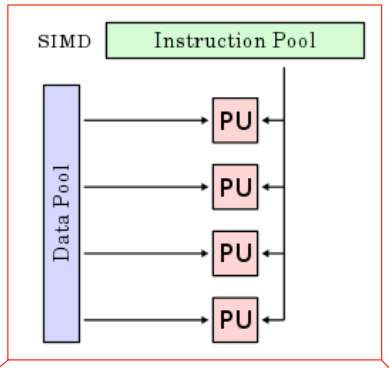
http://code.compeng.uni-frankfurt.de/projects/vc

Computer architectures

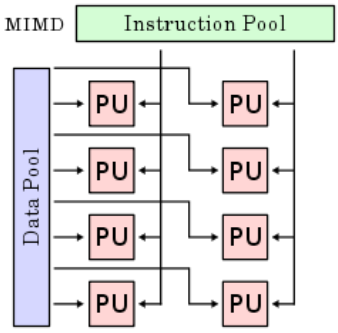
Single Instruction Single Data

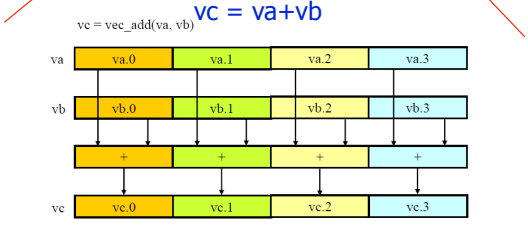


Single Instruction Multiple Data



Multiple Instruction Multiple Data





Taken from: http://en.wikipedia.org/wiki/Flynn's_taxonomy

Vc

- All functionality of SIMD headers
- Branch operators SIMDization using masks
- Random memory access using gather and scatter operators
- Many other additions

Masks

Scalar

```
float a(2);

   ...

if ( a < 0 )
  a = - a;

   ...

if (a > 2)
  return;
```

SIMDization

Vc

```
float_v a(2);

   ...

float_m mask = ( a < 0 );
a(mask) = - a;

   ...

float_m mask2 = ( a > 2 );
if ( mask2.isFull() )
  return;
```

Equivalent Scalar

```
float a[4] = {2,2,2,2};

   ...

if ( a[0] < 0 ) a[0] = - a[0];
if ( a[1] < 0 ) a[1] = - a[1];
if ( a[2] < 0 ) a[2] = - a[2];
if ( a[3] < 0 ) a[3] = - a[3];

   ...

if ( a[0] > 2 && a[1] > 2 &&
     a[2] > 2 && a[3] > 2)
  return;
```

Gather. Scatter

Scalar

```
float a[4], b[4];
float array[1000];

    …

a[0] = array[0];
a[1] = array[2];
a[2] = array[5];
a[3] = array[6];

    …

array[0] = b[0];
array[2] = b[1];
array[5] = b[2];
array[6] = b[3];
```

SIMDization

Vc

```
float_v a, b;
float array[1000];

    …

uint_v indexes;
  // fill indexes with {0,2,5,6}

a.gather( array, indexes );

    …

b.scatter( array, indexes );
```


Gather. Scatter

Scalar

```
struct MyData {
  float d1;
  int d2;
};

MyData array[1000]; // AoS
float a[4], b[4]; // SoA

    …

a[0] = array.d1[0];
a[1] = array.d1[2];
a[2] = array.d1[5];
a[3] = array.d1[6];

    …

array.d1[0] = b[0];
array.d1[2] = b[1];
array.d1[5] = b[2];
array.d1[6] = b[3];
```

Vc

```
struct MyData {
  float d1;
  int d2;
};

MyData array[1000];
float_v a, b;

    …

uint_v indexes;
  // fill indexes with {0,2,5,6}

a.gather( array, &MyData::d1, indexes );

    …

b.scatter( array, &MyData::d1, indexes );
```

Exercises

In order to make compiler know about Vc run:
. AddVcPath.sh
If Vc needs reinstallation follow instructions in Vc/readme.txt

Exersices/2_Vc/0_Memory

Task: Check out 2 ways to pack data in Vc

Exersices/2_Vc/1_Matrix

See the previous SIMD-exersices.
Task: Implement them with Vc

Exersices/2_Vc/2_QuadraticEquation

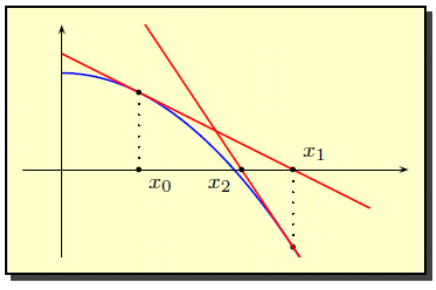
See the previous SIMD-exersices.
Task: Implement with Vc using three different initial data formats: AoS, SoA, AoSoA.

Exercises

Exersices/3_Vc/3_Newton

1. Newton's Method

Problem: Given an equation $f(x) = 0$, solve for x numerically.



- Make an *initial guess*: x_0. Now go up to the curve.
- Draw the tangent line. Its equation is

$$y = f(x_0) + f'(x_0)(x - x_0).$$

- Let x_1 be in x-intercept of this tangent line.

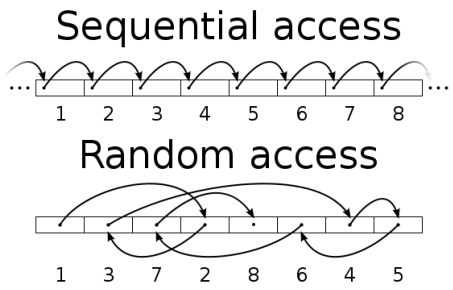
- This intercept is given by the formula: $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$.
- Now repeat using x_1 as the initial guess.
- The intercept x_2 is given by: $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$.
- Repeat until $(x_n - x_{n-1}) > \text{epsilon}$

Task: Vectorize using Vc

http://www.math.uakron.edu/~dpstory/tutorial/demos/newton.pdf

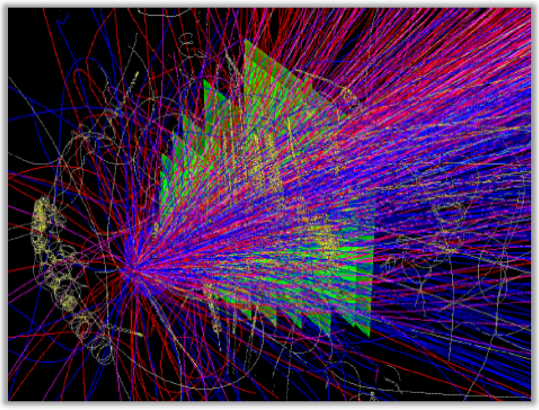
Exercises

Exersices/3_Vc/4_RandomAccess

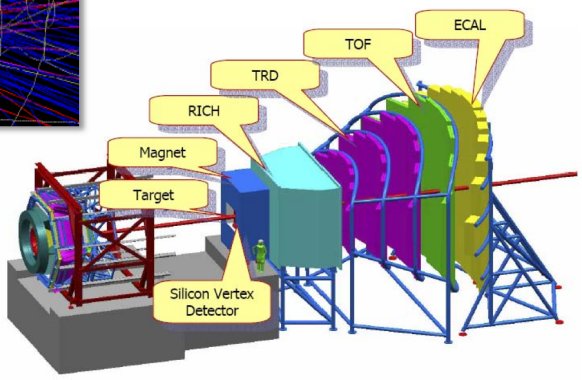


Task: Implement random access using gather and scatter operators.
Follow the directions in the comments.

Future CBM (FAIR/GSI) Experiment



- Fixed-target heavy-ion experiment
- 10^7 Au+Au collisions/s
- 1000 charged particles/collision
- Non-homogeneous magnetic field



No simple trigger primitive, like high p_t, available to tag events of interest. The only selective signature is the detection of the decay vertex.

Reconstruction packages:

• Track finding	Cellular Automaton (CA)
• Track fitting	Kalman Filter (KF)
• Vertexing	KFParticle

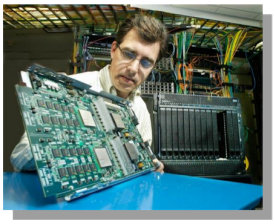
Full event reconstruction is required at the trigger level

CBM Kalman Filter Track Fit on Cell

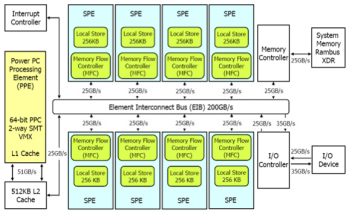
	Stage	Description	Time/track	Speedup
		Initial scalar version	12 ms	–
Intel P4	1	Approximation of the magnetic field	240 μs	50
Intel P4	2	Optimization of the algorithm	7.2 μs	35
Intel P4	3	Vectorization	1.6 μs	4.5
Cell	4	Porting to SPE	1.1 μs	1.5
Cell	5	Parallelization on 16 SPEs	0.1 μs	10
		Final simdized version	0.1 μs	120000

10000x faster on each CPU

Comp. Phys. Comm. 178 (2008) 374-383



The KF speed was increased by 5 orders of magnitude



blade11bc4 @IBM, Böblingen: 2 Cell Broadband Engines with 256 kB Local Store at 2.4 GHz

Motivated by, but not restricted to Cell !

Application of Kalman Filter to the CBM Track Fit

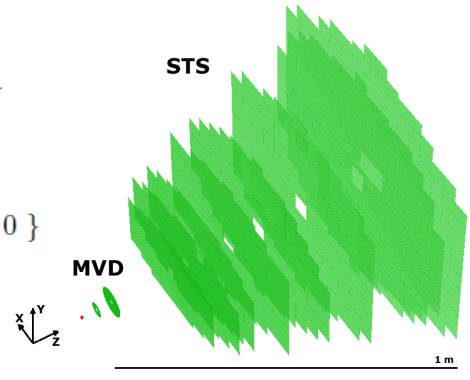
The state vector and its covariance matrix:

$$\mathbf{r} = \{\ x,\ y,\ t_x,\ t_y,\ q/p\ \}$$

Model of the measurement:

$$\mathbf{m}_k = H_k \mathbf{r}_k^t + \eta_k$$

$$H_k = \{\ cos(\alpha_k),\ sin(\alpha_k),\ 0,\ 0,\ 0\ \}$$



Prediction matrix:

$$F_{i,j} = \begin{pmatrix} 1 & 0 & \frac{\partial x^-}{\partial t_x^+} & \frac{\partial x^-}{\partial t_y^+} & \frac{\partial x^-}{\partial t_{\frac{q}{p}}^+} \\ 0 & 1 & \frac{\partial y^-}{\partial t_x^+} & \frac{\partial y^-}{\partial t_y^+} & \frac{\partial y^-}{\partial t_{\frac{q}{p}}^+} \\ 0 & 0 & \frac{\partial t_x^-}{\partial t_x^+} & \frac{\partial t_x^-}{\partial t_y^+} & \frac{\partial t_x^-}{\partial t_{\frac{q}{p}}^+} \\ 0 & 0 & \frac{\partial t_y^-}{\partial t_x^+} & \frac{\partial t_y^-}{\partial t_y^+} & \frac{\partial t_y^-}{\partial t_{\frac{q}{p}}^+} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$dt_x/dz = c\ t_r\ q/p\ (\ \ t_y(B_z + t_x B_x) - (1+t_x^2)B_y),$$
$$dt_y/dz = c\ t_r\ q/p\ (-t_x(B_z + t_y B_y) + (1+t_y^2)B_x),$$
$$t_r = \sqrt{t_x^2 + t_y^2 + 1},$$

Noise matrix:

$$Q_k = \sigma_k^2(\theta) \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & (t_x^2+1)t_r^2 & t_x t_y t_r^2 & 0 \\ 0 & 0 & t_x t_y t_r^2 & (t_y^2+1)t_r^2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\sigma(\theta) = \frac{13.6MeV}{\beta p} q\sqrt{S/X_0}\,[1 + 0.038\ln(S/X_0)]$$

Kalman Filter Track Fit for CBM experiment

Exersices/3_Vc/5_CBM_KF

- 3D
- Non-homogeneous magnetic field (sophisticated extrapolation)
- 5 tracks parameters: x, y, tx, ty, q/p
- 2D measurements (hits): x, y
- Vectorized using headers

Set up

ln -s ../../../../data data

Compile

make single (or "make singleVc")

Run

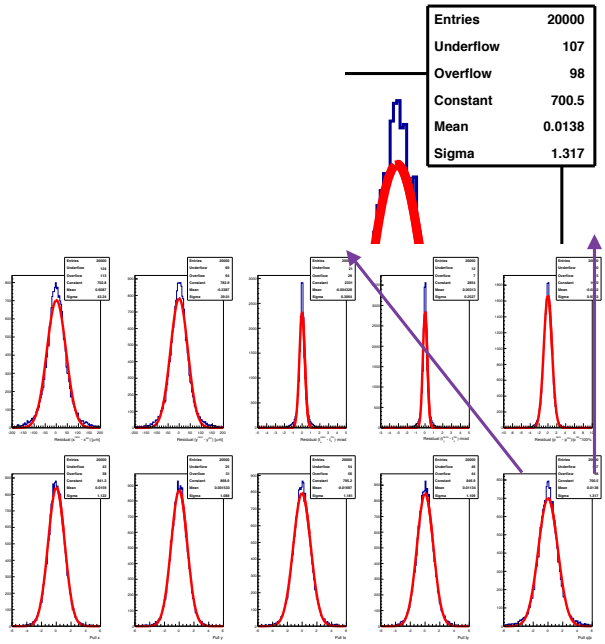
./single

Check results

cd QualityHisto

root –l –q histo_particle.C; root –l Pulls.C

(keep attention to sigma of distributions and number of entries)



Task:

Vectorize using Vc,

results should be the same within 0.1%

2.3. Vector classes Library

Exercises are located at Exercises/3_Vc/
Solutions are located at Exercises/3_Vc/***/***_solution.cpp
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

Vc Introduction

Vector classes (Vc) is a free software library to ease explicit vectorization of C++ code. It has an intuitive interface and provides portability between different compilers and compiler versions as well as portability between different vector instruction sets. Thus an application written with Vc can be compiled for AVX, SSE, XeonPhi (MIC) and others SIMD instructions[1].

Similar to the SIMD header files, it provides all basic arithmetic operations and functions, and much more in addition. All functionality, which the headers provide for **fvec** can be used similarly with the class **float_v**. The most useful functionalities of Vc are masks and random memory access.

The mask functionality allows conditioning calculations. For example, an absolute value of **a** can be calculated as:

float_m mask = (a < 0); a(mask) = - a;

Here **mask** saves the comparison result for each entry of **a** in a form, which can be represented like [true;true;false;true] and **operator()** applies the assignment only to those entries, where the mask is true.

The random memory access functionality is provided by the **gather** and **scatter** functions. For example:

gather fills a vector **a** from an array **A** taking elements with indexes stored in a vector **I**:

a.gather(A, I); // A[I] -> a

scatter makes the opposite - fills an array entries from a vector:

a.scatter(A, I); // a -> A[I]

3_Vc/0_Matrix: description

The Matrix exercise requires to parallelize the square root extraction over a set of float variables arranged in a square matrix using the Vc library. The initial code implements scalar and vector parts using the SIMD header, the space for the Vc implementation is blank. Therefore the initial output shows 0 time for vector calculations and infinite speed up factor, that should be currently ignored.

Part of the source code of Matrix.cpp

```
26  float a[N][N] __attribute__ ((aligned(16)));
27  float c[N][N] __attribute__ ((aligned(16)));
28  float c_simd[N][N] __attribute__ ((aligned(16)));
29  float c_simdVc[N][N] __attribute__ ((aligned(16)));
30
31  template<typename T> // required calculations
32  T f(T x) {
33    return sqrt(x);
34  }
…
49  int main() {
50
51     // fill classes by random numbers
52    for( int i = 0; i < N; i++ ) {
```

[1] http://code.compeng.uni-frankfurt.de/projects/vc

Part of the source code of Matrix.cpp

```
53      for( int j = 0; j < N; j++ ) {
54        a[i][j] = float(rand())/float(RAND_MAX); // put a random value, from 0 to
    1
55      }
56    }
57
58      /// -- CALCULATE --
59      /// SCALAR
60    TStopwatch timerScalar;
61    for( int ii = 0; ii < NIter; ii++ )
62      for( int i = 0; i < N; i++ ) {
63        for( int j = 0; j < N; j++ ) {
64          c[i][j] = f(a[i][j]);
65        }
66      }
67    timerScalar.Stop();
68
69      /// SIMD VECTORS
70    TStopwatch timerSIMD;
71    for( int ii = 0; ii < NIter; ii++ )
72      for( int i = 0; i < N; i++ ) {
73        for( int j = 0; j < N; j+=fvecLen ) {
74            fvec &aVec = (reinterpret_cast<fvec&>(a[i][j]));
75            fvec &cVec = (reinterpret_cast<fvec&>(c_simd[i][j]));
76            cVec = f(aVec);
77        }
78      }
79    timerSIMD.Stop();
80
81     /// Vc
82    TStopwatch timerVc;
83    //TODO write the code using Vc
84    timerVc.Stop();
```

Typical output

```
Time scalar:  798.745 ms
Time headers: 201.086 ms, speed up 3.97215
Time Vc:      0 ms, speed up inf
SIMD and scalar results are the same.
ERROR! SIMD and scalar results are not the same.
```

3_Vc/0_Matrix: solution

Since **float_v** is stored in memory in the same way as **fvec** and the function **f(...)** is a template, the part for Vc can be exactly the same, simply the name of the type must be changed:

Part of the source code of Matrix_solution.cpp

```
81      /// Vc
```

Part of the source code of Matrix_solution.cpp

```
82    TStopwatch timerVc;
83    for( int ii = 0; ii < NIter; ii++ )
84      for( int i = 0; i < N; i++ ) {
85        for( int j = 0; j < N; j+=float_v::Size ) {
86            float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
87            float_v &cVec = (reinterpret_cast<float_v&>(c_simd[i][j]));
88            cVec = f(aVec);
89        }
90      }
91    timerVc.Stop();
```

Typical output after solution

```
Time scalar:  798.728 ms
Time headers: 201.078 ms, speed up 3.97223
Time Vc:      201.079 ms, speed up 3.97221
SIMD and scalar results are the same.
SIMD and scalar results are the same.
```

Since 4 float variables fit into a single SIMD vector, all calculations are done in parallel and no overhead operations are required, the expected speed-up factor should be 4.

3_Vc/1_QuadraticEquation: description

The QuadraticEquation exercise requires to vectorize using Vc the solution of a set of quadratic equations in three different ways, based on tree different data setups: (1) Array of Structures (AoS), (2) Structure of Arrays (SoA) (3) Array of Structures of Arrays (AoSoA).

It is recommended to compile the code with **-fno-tree-vectorize** option to prevent auto-vectorization of the scalar code, otherwise the comparison of the vectorized and scalar codes will not be direct.

The input data is given already in the required formats. An elemental structure **DataAOSElement**, which contains parameters of the equations and the resulting maximum root, is declared for the AoS format. **_mm_malloc** function is used to allocate aligned memory.

Part of the source code of QuadraticEquation.cpp

```
23  struct DataAOSElement {
24    float a, b, c, // coefficients
25      x; // a root
26  };
27
28  struct DataAOS {
29    DataAOS(const int N) {
30      data = (DataAOSElement*) _mm_malloc(sizeof(DataAOSElement)*N,16);
31    }
32    ~DataAOS() {
33      if(data) _mm_free(data);
34    }
35    DataAOSElement *data;
36  };
```

The SoA format is declared as a whole structure, which contains dynamic arrays.

Part of the source code of QuadraticEquation.cpp

```
38 struct DataSOA {
39
40   DataSOA(const int N) {
41     a = (float*) _mm_malloc(sizeof(float)*N,16);
42     b = (float*) _mm_malloc(sizeof(float)*N,16);
43     c = (float*) _mm_malloc(sizeof(float)*N,16);
44     x = (float*) _mm_malloc(sizeof(float)*N,16);
45   }
46   ~DataSOA()
47   {
48     if(a) _mm_free(a);
49     if(b) _mm_free(b);
50     if(c) _mm_free(c);
51     if(x) _mm_free(x);
52   }
53
54   float *a, *b, *c, // coefficients
55     *x; // a root
56 };
```

To define the AoSoA format an elemental structure **DataAOSOAElement** is declared to contain information about **float_v::Size** (4) equations, similarly to **DataSOA**. The elemental structures are packed together in the DataAOSOA structure similarly to **DataAOS**. Memory for all information is allocated in one go in **DataAOSOA** to ensure compact data allocation, afterwards the memory is distributed between elemental structures using **SetMemory** function.

Part of the source code of QuadraticEquation.cpp

```
58 struct DataAOSOAElement {
59
60   void SetMemory(float *m) {
61     a = m;
62     b = m + float_v::Size;
63     c = m + 2*float_v::Size;
64     x = m + 3*float_v::Size;
65   }
66
67   float *a, *b, *c, // coefficients
68     *x; // a root
69 };
70
71 struct DataAOSOA {
72   DataAOSOA(const int N) {
73     const int NVectors = N/float_v::Size;
74
75     data = new DataAOSOAElement[NVectors];
76     memory = (float*) _mm_malloc(sizeof(float)*4*N,16);
77
78     float *memp = memory;
79     for( int i = 0; i < NVectors; i++ ) {
80       data[i].SetMemory(memp);
```

Part of the source code of QuadraticEquation.cpp

```
81        memp += float_v::Size*4;
82      }
83    }
84    ~DataAOSOA() {
85      _mm_free(memory);
86      delete[] data;
87    }
88
89    float *memory;
90    DataAOSOAElement *data;
91  };
```

The given structures is filled by the same random data sample and scalar implementation is given in the exercise code.

Part of the source code of QuadraticEquation.cpp

```
150   // fill parameters by random numbers
151   for( int i = 0; i < N; i++ ) {
152     float a = 0.01 + float(rand())/float(RAND_MAX); // put a random value, from
      0.01 to 1.01 (a has not to be equal 0)
153     float b = float(rand())/float(RAND_MAX);
154     float c = -float(rand())/float(RAND_MAX);
155
156     dataScalar.data[i].a = a;
157     dataScalar.data[i].b = b;
158     dataScalar.data[i].c = c;
159
160     dataSIMD1.data[i].a = a;
161     dataSIMD1.data[i].b = b;
162     dataSIMD1.data[i].c = c;
163
164     dataSIMD2.a[i] = a;
165     dataSIMD2.b[i] = b;
166     dataSIMD2.c[i] = c;
167
168     const int nV = i/float_v::Size;
169     const int iV = i%float_v::Size;
170     dataSIMD3.data[nV].a[iV] = a;
171     dataSIMD3.data[nV].b[iV] = b;
172     dataSIMD3.data[nV].c[iV] = c;
173   }
174
175   /// -- CALCULATE --
176
177   // scalar calculations
178   TStopwatch timerScalar;
179   for(int io=0; io<NIterOut; io++)
180     for(int i=0; i<N; i++)
181     {
182       float &a = dataScalar.data[i].a;
```

Part of the source code of QuadraticEquation.cpp

```
183       float &b = dataScalar.data[i].b;
184       float &c = dataScalar.data[i].c;
185       float &x = dataScalar.data[i].x;
186
187       float det = b*b - 4*a*c;
188       x = (-b+sqrt(det))/(2*a);
189     }
190   timerScalar.Stop();
```

Typical output

```
Time scalar:   567.055 ms
Time Vc AOS:   0 ms, speed up inf
Time Vc SOA:   0 ms, speed up inf
Time Vc AOSOA: 0 ms, speed up inf
ERROR! SIMD and scalar results are not the same. SIMD part 1.
ERROR! SIMD and scalar results are not the same. SIMD part 2.
ERROR! SIMD and scalar results are not the same. SIMD part 3.
```

3_Vc/1_QuadraticEquation: solution

(1) To vectorize the AoS format one needs to gather data from different instances of the **DataAOSElement** structure and pack it together (in groups by 4) in **float_v** variables. Since the same data (for example, the **a** variable) is placed in different parts of memory, data coping is necessary here. Ones data is grouped, the solution is found using exactly the same code as one given in the scalar part. The data is ungrouped back into the output **x** variable.

Part of the source code of QuadraticEquation_solution.cpp

```
194   for(int io=0; io<NIterOut; io++)
195   {
196     for(int i=0; i<NVectors; i++)
197     {
198       // copy input data
199       float_v aV;
200       float_v bV;
201       float_v cV;
202
203
204       for(int iV=0; iV<float_v::Size; iV++)
205       {
206         aV[iV] = dataSIMD1.data[i*float_v::Size + iV].a;
207         bV[iV] = dataSIMD1.data[i*float_v::Size + iV].b;
208         cV[iV] = dataSIMD1.data[i*float_v::Size + iV].c;
209       }
210
211       const float_v det = bV*bV - 4*aV*cV;
212       float_v xV = (-bV+sqrt(det))/(2*aV);
213
214       // copy output data
215       for(int iE=0; iE<float_v::Size; iE++)
```

Part of the source code of QuadraticEquation_solution.cpp

```
216         dataSIMD1.data[i*float_v::Size+iE].x = xV[iE];
217       }
218     }
```

(2) The second task is vectorization with the SoA data format. Since in SoA similar data is placed near each other, the **reinterpret_cast** can be used. Ones the data is reinterpreted, the solution is found using exactly the same code, as the given one in the scalar part.

Part of the source code of QuadraticEquation_solution.cpp

```
223   for(int io=0; io<NIterOut; io++)
224     for(int i=0; i<N; i+=float_v::Size)
225     {
226       float_v& aV = (reinterpret_cast<float_v&>(dataSIMD2.a[i]));
227       float_v& bV = (reinterpret_cast<float_v&>(dataSIMD2.b[i]));
228       float_v& cV = (reinterpret_cast<float_v&>(dataSIMD2.c[i]));
229       float_v& xV = (reinterpret_cast<float_v&>(dataSIMD2.x[i]));
230
231       const float_v det = bV*bV - 4*aV*cV;
232       xV = (-bV+sqrt(det))/(2*aV);
233     }
```

(3) The third task is vectorization with the AoSoA data format. The reinterpret cast can be used here in the same way as with the SoA format, just the dereference operator must be applied in addition. The calculations part remains the same as well.

Part of the source code of QuadraticEquation_solution.cpp

```
238   for(int io=0; io<NIterOut; io++)
239     for(int i=0; i<NVectors; i++)
240     {
241       float_v& aV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].a));
242       float_v& bV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].b));
243       float_v& cV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].c));
244       float_v& xV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].x));
245
246       const float_v det = bV*bV - 4*aV*cV;
247       xV = (-bV+sqrt(det))/(2*aV);
248     }
```

Typical output after solution

```
Time scalar:   566.821 ms
Time Vc AOS:   217.513 ms, speed up 2.60592
Time Vc SOA:   142.603 ms, speed up 3.97482
Time Vc AOSOA: 143.57 ms, speed up 3.94805
SIMD 1 and scalar results are the same.
SIMD 2 and scalar results are the same.
SIMD 3 and scalar results are the same.
```

The speed up factor of 4 is expected due to vectorization. It is achieved with the SoA and AoSoA data formats. With AoS additional data regrouping is required, that results in the smaller speed up of 2.6.

3_Vc/2_Newton: description

The Newton exercise requires to vectorize the Newton method for numerical solution of a group of equations.

The method can be explained graphically as in the Fig 1. The task is to find intersection of the curve and the X-axis. The algorithm starts with an initial approximation for root x_0 and takes a function value at this point. A tangent line is drown at this point, an intersection of the tangent line with the X-axis is the next approximation x_1:

$x_k \equiv x_{k-1} - f(x_{k-1})/f'(x_{k-1})$

Then the procedure is repeated until the approximation does not change within the required precision.

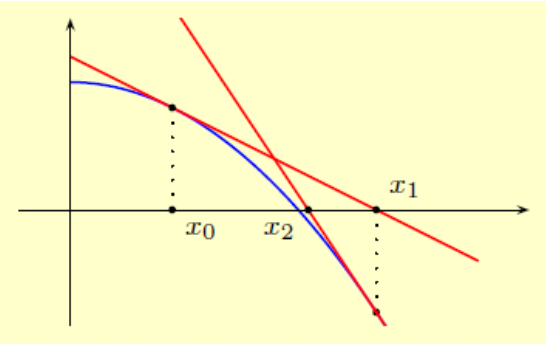


Fig. 1. Explanation of the Newton task.

$x_k - x_{k-1} < epsilon$

A scalar version of the algorithm is given in the exercise. A solution is proposed to perform in two steps. First, vectorize the algorithm, which uses a fixed number of iterations (1000). Then make the number of iterations dependent on the required precision. The vectorized version must give exactly the same result as the scalar one.

Part of the source code of Newton.cpp

```
44 float FindRootScalar(const float& p1, const float& p2)
45 {
46   float x = 1, x_new = 0;
47   for( ; abs((x_new - x)/x_new) > P; ) {
48       // for(int i = 0; i < 1000; ++i){
49     x = x_new;
50     x_new = x - F(x,p1,p2) / Fd(x,p1,p2);
51   }
52   return x_new;
```

Typical output

```
Scalar part:
Results are correct!
Time: 15.415 ms
SIMD part:
Results are NOT the same!
Time: 0.226021 ms
Speed up: 68.2015
```

3_Vc/2_Newton: solution

The parallelization is achieved by grouping 4 equations together. A complication in this case is that different equations will reach the required precision at different number of iterations. This problem can be solved using Vc masks: the loop exit condition should be stored as a mask. Then the loop is continued until the mask has at least one entry with the value **true**. A mask value should be also used during update of the root approximation in order to reproduce the scalar result.

Part of the source code of Newton_solution.cpp

```
56 float_v FindRootVector(const float_v& p1, const float_v& p2)
57 {
58   float_v x = 1.f, x_new = 0.f;
59   float_m mask(true);
60   for( ; !mask.isEmpty(); ) {
61       // for(int i = 0; i < 1000; ++i){
62     x = x_new;
63     x_new(mask) = x - F(x,p1,p2) / Fd(x,p1,p2);
64     mask = abs((x_new - x)/x_new) > P;
65   }
66   return x_new;
67 }
```

Typical output after solution

```
Scalar part:
Results are correct!
Time: 15.3401 ms
SIMD part:
Results are the same!
Time: 4.49395 ms
Speed up: 3.4135
```

A typical speed up factor of vectorization is 4, but since mask is used, the parallelization is not full and at the final iterations only a part of the SIMD vector entries is used. Therefore, the speed up factor is about 3.5.

3_Vc/3_RandomAccess: description

The RandomAccess exercise requires to use different forms of the **gather** and **scatter** functions. For that an input array of **float** is provided and randomly filled. Also an array of random indices is provided. It is required to (1) gather data from the **input** array to the tmp **float_v** variable according to the **index** array; (2) similarly, gather the data, but only when it satisfies a given condition; (3) put the data from the tmp **float_v** variable to the **output** array, when it satisfies a given condition.

Part of the source code of RandomAccess.cpp

```
21 float input[N];
22 float output[N];
23
24 int main() {
25
26   unsigned int index[float_v::Size];
27
30   // fill input array with random numbers from 0 to 1
31   for( int i = 0; i < N; i++ ) {
32     input[i] = float(rand())/float(RAND_MAX);
33   }
34
35   // fill output array with 0
36   for( int i = 0; i < N; i++ ) {
37     output[i] = 0;
38   }
```

Part of the source code of RandomAccess.cpp

```
39
40    // fill indices with random numbers from 0 to N-1
41    for( int i = 0; i < float_v::Size; i++) {
42      index[i] = static_cast<unsigned int>(float(rand())/float(RAND_MAX)*N);
43    }
44
45    cout << "Indices: ";
46    for(int i=0; i<float_v::Size; i++)
47      cout << index[i] << " ";
48    cout << endl;
49
50      /// gather without masking
51    float_v tmp;
52    //TODO gather data with indices "index" from the array "input" into float_v
      tmp
53    // Use void  gather (const float *array, const uint_v &indexes)
…
65      /// gather with masking
66    float_v tmp2;
67    //TODO gather data with indices "index" from the array "input" into float_v
      tmp2, if the value of "input" large then 0.5
68    // Use void  gather (const float *array, const uint_v &indexes, const float_m
      &mask)
…
91    //TODO create mask for values for an obtained tmp values, which are large
      than 0.5 and
92    //TODO put all values smaller than 0.5 from tmp to the array "output" at the
      places given by indices "index"
93    // Use void scatter (float *array, const uint_v &indexes, const float_m
      &mask) const
```

Typical output

```
Indices: 25 96 76 42
Gather without masking: results are WRONG.
-4.67253e+33 is not equal to 0.988475
4.59163e-41 is not equal to 0.912037
0 is not equal to 0.80679
Gather with masking: results are WRONG.
Scatter with masking: results are correct.
```

3_Vc/3_RandomAccess: solution

(1) To gather data from the given places in an array, one needs to create a **uint_v** vector with corresponding indexes, then the **gather** function can be applied directly.

Part of the source code of RandomAccess_solution.cpp

```
53    uint_v ind(index);
54    tmp.gather(input,ind);
```

(2) To gather data under the given condition, one needs to create a mask with a corresponding type. Since the data is **float**, the **float_m** mask must be created, then the **gather** function with the mask

parameter is applied directly. To ensure that the entries, which were masked out, have meaningful values, we need to initialise the output variable before gathering.

Part of the source code of RandomAccess_solution.cpp

```
67    float_m mask = tmp > 0.5f;
68    float_v tmp2(Vc::Zero);
69    tmp2.gather(input, ind, mask);
```

(3) To scatter data under the given condition one needs to create a mask with a corresponding type. Since the data is **float**, the same mask can be used as in (2), then the **scatter** function with the mask parameter is applied directly.

Part of the source code of RandomAccess_solution.cpp

```
93    mask = tmp < 0.5f;
94    tmp.scatter(output, ind, mask);
```

Typical output after solution

```
Indices: 98 40 24 14
Gather without masking: results are correct.
Gather with masking: results are correct.
Scatter with masking: results are correct.
```

3_Vc/5_CBM_KF: description

For the next vectorisation exercise we will have a closer look at Kalman filter (KF) algorithm in the case of 3 dimensional task of tracks reconstruction in magnetic field. The CBM KF package version, which is already vectorized using headers is given. The task is to implement the CBM KF with Vc instead of headers.

The related to this exercises parts of the package are: **Fit.cxx; FitClasses.h; Fit.h; Stopwatch.h; Makefile** and **openlab_mod** directory, and **QualityHisto** directory.

Fit.cxx file contains the main function and other high-level functions are described here, including **ReadInput, FitTracksV** and **WriteOutput**. **ReadInput** and **WriteOutput** functions read and write data, which is at the **data** directory. **FitTracksV** is the function, which packs data into **fvec**-variables and implements the KF method.

Part of the source code of Fit.cxx

```
1     #include <iostream>
2     #include <fstream>
…
11    #include "Stopwatch.h"
12    #include "Fit.h"
…
60    Station* vStations;
61
62    Track vTracks[MaxNTracks];
63    MCTrack vMCTracks[MaxNTracks];
…
72    void ReadInput(){
73
74      fstream FileGeo, FileTracks;
75
76      FileGeo.open( (dataDir+"geo.dat").c_str(), ios::in );
77      FileTracks.open( (dataDir+"tracks.dat").c_str(), ios::in );
```

Part of the source code of Fit.cxx

```
…
178 }
179
180 void WriteOutput(){
…
233 }
234
235
236 void FitTracksV(){
237
238   double TimeTable[Ntimes];
239
240   TrackV *TracksV = new TrackV[MaxNTracks / vecN + 1];
241   Fvec_t *Z0      = new Fvec_t[MaxNTracks/vecN+1];
242
243 #ifdef VC
244   cout << " VC code is not writen yet " << endl; // DELME
245   exit(1);
246     // TODO
247 #endif
248 #ifndef MUTE
249   cout<<"Prepare data..."<<endl;
250 #endif
251   Stopwatch timer1;
252
253   for( int iV=0; iV<NTracksV; iV++ ){ // loop on set of 4 tracks
254 #ifndef MUTE
255     if( iV*vecN%100==0 ) cout<<iV*vecN<<endl;
256 #endif
257     TrackV &t = TracksV[iV];
258     for( int ist=0; ist<NStations; ist++ ){
259       HitV &h = t.vHits[ist];
260
261       h.x = 0.;
262       h.y = 0.;
263       h.w = 0.;
264       h.H.X = 0.;
265       h.H.Y = 0.;
266       h.H.Z = 0.;
267     }
268
269     for( int it=0; it<vecN; it++ ){
270       Track &ts = vTracks[iV*vecN+it];
271 #ifdef X87
272       Z0[iV] = vMCTracks[iV*vecN+it].MC_z;
273 #else
274       Z0[iV][it] = vMCTracks[iV*vecN+it].MC_z;
275 #endif
276       for( int ih=0; ih<ts.NHits; ih++ ){
277         Hit &hs = ts.vHits[ih];
278         HitV &h = t.vHits[hs.ista];
279     #ifdef X87
280         h.x = hs.x;
281         h.y = hs.y;
```

Part of the source code of Fit.cxx

```
282         h.w = 1.;
283     #else
284         h.x[it] = hs.x;
285         h.y[it] = hs.y;
286         h.w[it] = 1.;
287     #endif
288       }
289     }
…
346 #pragma omp parallel num_threads(tasks)
347   {
348     #pragma omp for
349       for( iV=0; iV<NTracksV; iV++ ){ // loop on set of 4 tracks
350 // timer_test.Start();
351         for( ifit=0; ifit<NFits; ifit++){
352           Fit( TracksV[iV], vStations, NStations );
353          }
354 // timer_test.Stop();
355 // cout<<"test time = "<<timer_test.RealTime()*1.e6<<" [us]"<<endl;
356       }
357   }
…
416 }
417
418
419 int main(int argc, char *argv[]){
..
445   ReadInput();
446   FitTracksV();
447   WriteOutput();
…
459 }
460
```

FitClasses.h file contains description of the objects used during fitting, including definition of basics types, which are used depending on the switcher.

There are number of switcher options (targets), useful for the given task are: **X87**, which corresponds to the scalar version of the code, and **SIMPLESIMD**, which corresponds to the vectorized version using SIMD header. The basic type, which is used for calculations, is **Fvec_t**, it's length is stored in **vecN** variable. Therefore for **X87 Fvec_t** is set to double, **vecN** to 1, for SIMD header it is set to **F32vec4** (the different name of **fvec**), **vecN** to 4.

The objects, which are used during track fit, are:

FieldVector - contains magnetic field components at the given point.

FieldSlice - contains magnetic field approximation at the given detector station plane.

FieldRegion - contains magnetic field approximation along the given curve (parabola, which is defined by 3 points).

Station - contains detector plane parameters and magnetic field on the detector plane.

Hit, HitV - scalar and vector hits, they contains hit coordinates and magnetic field in the hit position.

MCTrack - contains information about simulated tracks to compare resulting reconstructed track approximation with.

Track, TrackV - scalar and vectorized track, they contains hits and parameters of the track, obtained by the Kalman filter.

CovV - contains covariance matrix of the estimated track's parameters.

Part of the source code of FitClasses.h

```
…
#elif defined( X87 )

#include "openlab_mod/x87.h"
typedef double Fvec_t;
const int vecN = 1;
typedef double  Single_t;
# define ALIGNMENT_NONE
…
#elif defined( SIMPLESIMD )

#include "openlab_mod/P4_F32vec4.h"
typedef F32vec4 Fvec_t;
const int vecN = 4;
typedef float  Single_t;
# define ALIGNMENT_16
…
#endif

typedef int    Int_t;

struct FieldVector{
  Fvec_t X, Y, Z;
…
} __attribute__ ((aligned(16)));
…
struct FieldSlice{
  Fvec_t X[10], Y[10], Z[10]; // polinom coeff.
…
} __attribute__ ((aligned(16)));


struct FieldRegion{
  Fvec_t x0, x1, x2 ; // Hx(Z) = x0 + x1*(Z-z) + x2*(Z-z)^2
  Fvec_t y0, y1, y2 ; // Hy(Z) = y0 + y1*(Z-z) + y2*(Z-z)^2
  Fvec_t z0, z1, z2 ; // Hz(Z) = z0 + z1*(Z-z) + z2*(Z-z)^2
  Fvec_t z;
…
} __attribute__ ((aligned(16)));


struct Station{
  Fvec_t z, thick, zhit, RL,  RadThick, logRadThick,
    Sigma, Sigma2, Sy;
  FieldSlice Map;
…
} __attribute__ ((aligned(16)));


struct Hit{
  Single_t x, y;
  Int_t ista;
  Single_t tmp1;
```

Part of the source code of FitClasses.h

```
174 } __attribute__ ((aligned(16)));
175
176 struct MCTrack{
177   Single_t MC_x, MC_y, MC_z, MC_px, MC_py, MC_pz, MC_q;
178 } __attribute__ ((aligned(16)));
179
180 struct Track{
181   Int_t NHits;
182   Hit vHits[12];
183   Single_t T[6]; // x, y, tx, ty, qp, z
184   Single_t C[15]; // cov matr.
185   Single_t Chi2;
186   Int_t NDF;
187 } __attribute__ ((aligned(16)));
188
189 struct HitV{
190   Fvec_t x, y, w;
191   FieldVector H;
192 } __attribute__ ((aligned(16)));
193
194
195 struct CovV{
196   Fvec_t C00,
197     C10, C11,
198     C20, C21, C22,
199     C30, C31, C32, C33,
200     C40, C41, C42, C43, C44;
201 } __attribute__ ((aligned(16)));
202
203 struct TrackV{
204   HitV vHits[12];
205   Fvec_t T[6]; // x, y, tx, ty, qp, z
206   CovV   C;    // cov matr.
207   Fvec_t Chi2;
208   Fvec_t NDF;
209 } __attribute__ ((aligned(16)));
210
211 #endif
```

Fit.h contains description of the KF functions used for parameters estimation:
ExtrapolateALight - is used for extrapolation of track parameters from one station to the next one.
Filter - is used to take into account a given hit.
FilterFirst - is used to take into account (filter) the very first hit of the track. This function is equivalent to **Filter**, but optimised to cope with a numerical instability of the method when using single precision. **Filter** function is not stable when applied to the very first hit. The reason is big round-off errors when initial imprecise approximation of track parameters is combined with precise measurement.
AddMaterial - is used to take into account multiple scattering in material of the detector, that tracks are crossing.
GuessVec - is used to obtain approximation for the initial track parameters. This function also required to increase stability of the Kalman filter procedure in single precision.
The **Fit** function run the track fit process itself calling the other functions in a loop over stations.

Part of the source code of Fit.h

```
4 #include <math.h>
```

Part of the source code of Fit.h

```
5   #include "FitClasses.h"
…
16  //inline // --> causes a runtime overhead and problems for the MS compiler
    (error C2603)
17  void ExtrapolateALight
18  (
19   Fvec_t T [], // input track parameters (x,y,tx,ty,Q/p)
20   CovV &C,     // input covariance matrix
21   const Fvec_t &z_out  , // extrapolate to this z position
22   Fvec_t       &qp0    , // use Q/p linearisation at this value
23   FieldRegion &F
24   )
25  {
…
301 }
302
303 struct HitInfo{
304   Fvec_t cos_phi, sin_phi, sigma2, sigma216;
305 };
306
307 inline void Filter( TrackV &track, HitInfo &info, Fvec_t &u, Fvec_t &w )
…
380 inline void FilterFirst( TrackV &track, HitV &hit, Station &st )
…
399 inline void AddMaterial( TrackV &track, Station &st, Fvec_t &qp0 )
…
425 inline void GuessVec( TrackV &t, Station *vStations, int NStations )
…
490 inline void Fit( TrackV &t, Station vStations[], int NStations )
491 {
…
504   GuessVec( t, vStations,NStations );
…
516   FilterFirst( t, *h, vStations[i] );
517   AddMaterial( t, vStations[ i ], qp0 );
…
529   for( --i; i>=0; i-- ){
…
538     ExtrapolateALight( t.T, t.C, st.zhit, qp0, f );
540
541     AddMaterial( t, st, qp0 );
542
543     Filter( t, Xinfo, h->x, h->w );
544     Filter( t, Yinfo, h->y, h->w );
…
549   }
550 }
```

Stopwatch.h file is used to calculate a fitting time.

Makefile is required to organise compilation of several source files into one executable. It also allows to chose the target (for example, to choose between scalar version, or SIMD header version or Vc version). The compilation of the package is done with **make** command, which uses **Makefile** for directions.

The **Makefile** contains description of compiler options, the list of source files and the list of targeting executables. **singeVc** executable is already added here to be used with Vc, it defines the switcher option **VC**, which should be used in the C++ source code to distinguish the code version, which corresponds to the Vc target.

Part of the source code of Makefile

```
…
17  CFLAGS        = -pipe -Wall -W -D_REENTRANT $(DEFINES)
18  CXXFLAGS      = -msse -msse2 -msse3 -mssse3 -O3 -Wno-long-long -Iopenlab_mod -
    fno-threadsafe-statics
…
48  SOURCES       = Fit.cxx Fit.h FitClasses.h
50
51  BINARIES      = single singleVc tbb tbbVc omp ompVc pseudo x87 double
…
64  single: openlab_mod/P4_F32vec4.h $(SOURCES)
65    $(CXX) -DSIMPLESIMD $(CXXFLAGSSIMPL) $(LIBS) -o "$@" Fit.cxx
…
77  singleVc: $(SOURCES)


…
87  x87: $(SOURCES)
88    $(CXX) -DVC $(CXXFLAGS) $(LIBS) -o "$@" Fit.cxx  -lVc
…
```

To compile package with the **Makefile** one simply runs:

make [target name]

where **[target name]** is the name one of the targets, for example: **make single**.
To run the program call the executable name (same as target) as usually:

./single

openlab_mod directory contains SIMD header files, used for vectorization.

QualityHisto directory contains scripts, which allows to check result of the fitting procedure. **histo_particle.C** macros is used to analyse the package output and store the data in a root-file as a histograms (the output file is **histo_particle.root**). **Pulls.C** macros is used to open the root-file, plot the histograms on the screen and fit them with the Gaussian functions to find out the main characteristics of the fitted distributions: widths of resolutions and pulls. To perform described analysis one simply executes:

root -l histo_particle.C+ -q; root -l Pulls.C

The resulting picture for single target is shown on Fig. 1. It is also given in **QualityHisto/ KFTrackPull_single_etalon.pdf** file. It presents residual, resolution and pulls distributions. Residual is defined as difference between reconstructed and MC parameter values. Resolution is residual in [%] units. Pulls are residuals normalised on the errors obtained by the KF procedure (the errors are stored in the covariance matrix). Residuals are shown for x, y, tx, ty parameters. Since all tracks have the same length and almost same conditions (field is smooth, material of the stations is homogenious) the shape of residuals should be close to the Gaussian function given by red line (approximation), the width of the function represents the quality of the fitting method. For example, the **x** coordinate has the **Sigma** parameter equal to 43.24 microns, which is a precision of estimation of the **x** coordinate of the track. For the momentum **p** the resolution is plotted. Pulls are show for all 5 parameters. Since pulls are residuals normalised by the error, the width of pulls distributions should be equal to 1. It is close to one as one can see. The biggest difference is obtained for the pull of momentum (q/p). The explanation for such a

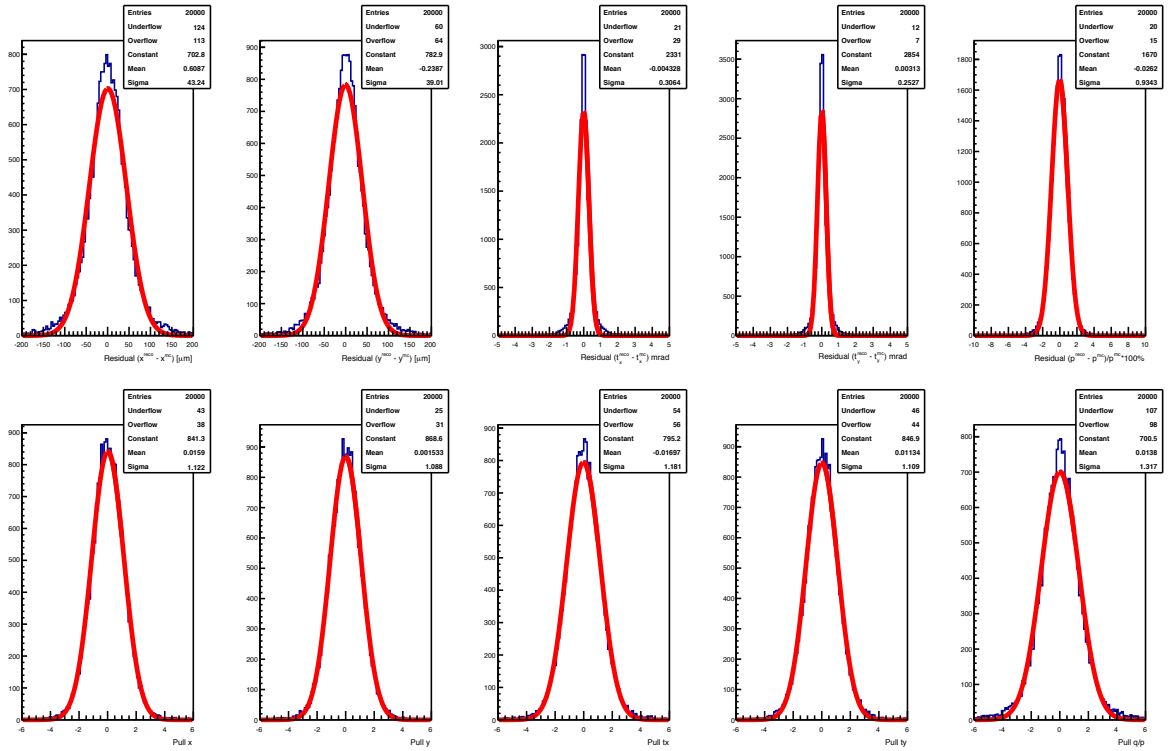


Fig. 1. Quality histograms for CBM KF track fitter. SIMD header version.

difference is approximations, which had to be done in order to perform Kalman filter procedure in a non-homogeneous magnetic field.

The most important parameters here are Sigmas and number of entries (see Fig. 2.). During solution it is proposed to use for benchmark Sigma and Entries of p resolution and q/p pull distribution, they are equal to 0.93 43, 20000, 1.317 and 20000 respectively. Typical time is about 0.5 microseconds per track (see Real fit/tr[us] time).

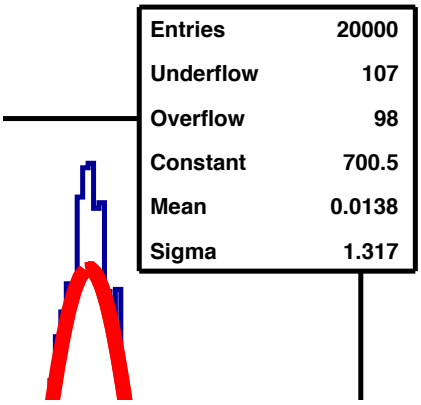


Fig. 2. Important parameters of the histograms for CBM KF track fitter.

```
Typical output
Prep[us], CPU fit/tr[us], Real fit/tr[us], CPU[sec], Real[sec] = 0.5    0.575
0.579457    1.15  1.15897
```

2_Vc/5_CBM_KF: solution

Since **Fvec_t** is the basic type used through the program, the solution of this exercise should start from redefining this type for the Vc version. As it was shown in description of the package such a definitions are done at **FitClasses.h**. All required definitions for the Vc version we collected in a separate file **Vctypedef.h**.

It contains redefinition of **Fvec_t**, which is clearly should be **float_v** in our case, **vecN**, which is equal to **float_v::Size** and redefinition of the input operator for a convenience. In addition **rcp** function redefinition is required, the Vc function, that corresponds to it is called **reciprocal**.

Part of the source code of Fit.h

```
54 #elif defined( VC )
55
56 #include "Vctypedef.h"
57 typedef float Single_t;
58
59 #endif
```

Part of the source code of Vctypedef.h

```
4  #include <Vc/Vc>
5  using namespace Vc;
6  typedef float_v Fvec_t;
7  const int vecN = float_v::Size;
8
9  istream & operator>>(istream &strm, Fvec_t &a ){
10   float tmp;
11   strm>>tmp;
12   a = tmp;
13   return strm;
14 }
15
16 inline Fvec_t rcp(const Fvec_t &a) {
17   return reciprocal(a);
18 }
```

Vc requires explicit type for floating point constants, therefore everywhere where constant of **Fvec_t** type are used, like 1 or 2, ".f" construct must be added to them. For example "1.f".

The last change to make Vc version compiled is implementation of data packing. It can be added right in the place where input data is read. **VC** switch is used to separate new code from the previous, this allows to compile the same code both for Vc and SIMDheader. To pack data into Vc classes aligned arrays of floats are used, then constructor function of **float_v** type is called to load data from given memory to the Vc type.

Part of the source code of Fit.cxx

```
244 #ifdef VC
245   float Z0mem[vecN] __attribute__ ((aligned(16)));;
246 #endif
…
268 #ifdef VC
269     float hxmem[NStations][vecN],hymem[NStations][vecN],hwmem[NStations][vecN]
__attribute__ ((aligned(16)));;
270     for( int it=0; it<vecN; it++ ){
271       Track &ts = vTracks[iV*vecN+it];
272
273       Z0mem[it] = vMCTracks[iV*vecN+it].MC_z;
274
275       for( int ista=0, ih=0; ista<NStations; ista++ ){
276         Hit &hs = ts.vHits[ih];
277         if (hs.ista != ista) continue;
```

Part of the source code of Fit.cxx

```
278         ih++;
279         HitV &h = t.vHits[hs.ista];
280
281         hxmem[ista][it] = hs.x;
282         hymem[ista][it] = hs.y;
283         hwmem[ista][it] = 1.;
284       }
285
286     }
287     for( int ista=0; ista<NStations; ista++ ){
288       Fvec_t hxtemp( hxmem[ista] );
289       Fvec_t hytemp( hymem[ista] );
290       Fvec_t hwtemp( hwmem[ista] );
291       t.vHits[ista].x = hxtemp;
292       t.vHits[ista].y = hytemp;
293       t.vHits[ista].w = hwtemp;
294     }
295
296
297     Fvec_t Z0temp(Z0mem);
298     Z0[iV] = Z0temp;
299 #else // VC
```

The data unpacking is already done in SIMDheader version, the same code can be used for Vc too, since access operator **operator[]** exists for both. Therefore no code changes are required here.

The result is shown on Fig. 3. It is clear that difference with header version is huge.

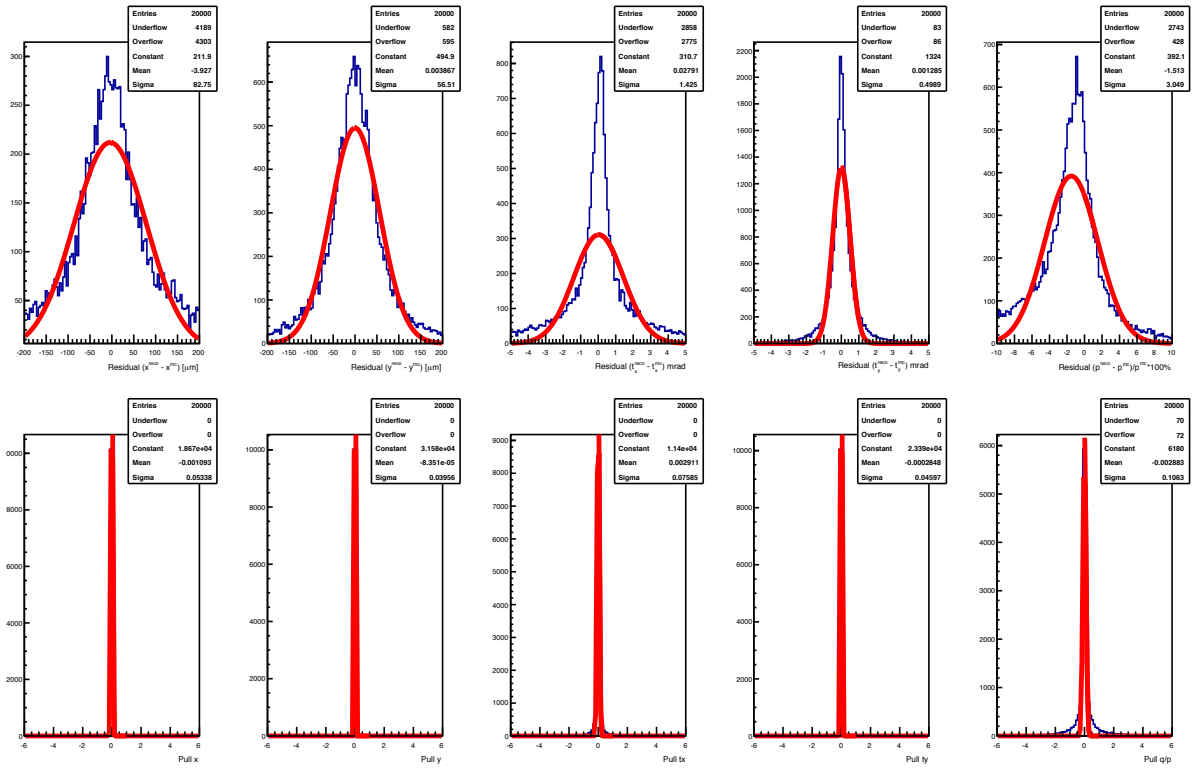


Fig. 3. Quality histograms for CBM KF track fitter. First Vc version.

The version is compilable, but the code is still must be checked to be sure that change from **fvec** to **float_v** is correctly done everywhere. Since code is mostly have no branches this is mostly true, but one part where **operator&** is used with SIMD header to achieve effect of conditional operator. That the conditioning must be used is clear from the scalar code. But **operator&** behaviour is not defined for **float_v** the concept for conditioning execution in Vc is masking. Therefore instead of **bool initialised** the mask should be used.

Part of the source code of Fit.h

```
337 #ifdef X87
338   bool initialised = HCH < info.sigma216;
339   if (initialised) {
340     track.Chi2 += zeta * zetawi;
341     zetawi = w* zeta *rcp(info.sigma2 + HCH);
342   } else
343     zetawi = w* zeta *rcp(HCH);
344 #elif defined(VC)
345   float_m initialised = HCH < info.sigma216;
346   wi = w*1/(info.sigma2 +HCH);
347   Fvec_t sigma(Vc::Zero);
348   sigma(initialised) = info.sigma2;
349   zetawi = w * zeta * 1/( sigma + HCH );
350   track.Chi2(initialised) += zeta * zetawi;
351 #else
352   Fvec_t initialised = Fvec_t( HCH<info.sigma216 );
353   zetawi = w* zeta *rcp( (initialised&info.sigma2) + HCH);
354   track.Chi2 += initialised & (zeta * zetawi);
355 #endif
```

The results are shown in Fig. 4. Benchmarking Sigma and Entries of p resolution and q/p pull distribution values are equal to 0.9341, 20000, 1.316 and 20000 respectively. The results difference with header version is less than 0.1% in Sigma and is 0 in number of entries, which is totally fine.

The time results are quite the same as with the SIMD header version. Vc can be slightly faster, because it uses time optimised functions like **reciprocal** and **rsqrt**.

Typical output

```
Prep[us], CPU fit/tr[us], Real fit/tr[us], CPU[sec], Real[sec] = 0.5    0.575
0.579457    1.15  1.15897
```

Typical output after solution

```
Prep[us], CPU fit/tr[us], Real fit/tr[us], CPU[sec], Real[sec] = 0.5    0.57
0.569018    1.14  1.13808
```

In addition we can check our guess about **reciprocal** and **rsqrt** functions. When we change then reciprocal on 1/x, which it's definition and rsqrt on 1/sqrt. For this version the benchmarking Sigma and Entries of p resolution and q/p pull distribution values are equal to 0.934, 20000, 1.317 and 20000 respectively. This makes the results difference with header version ~0.01% in Sigma, which is negligible.

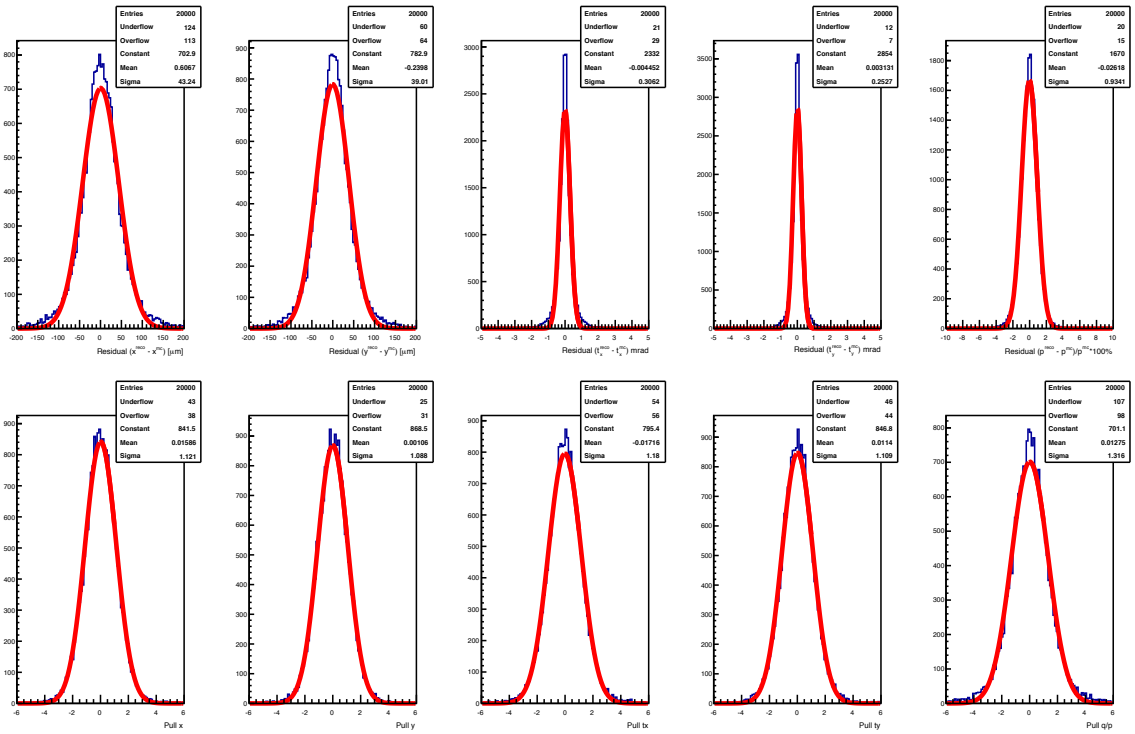


Fig. 4. Quality histograms for CBM KF track fitter. Vc version.

HPC Practical Course
Part 2.4

CERN ROOT Framework

V. Akishina, I. Kisel
I. Kulakov, M. Zyzak

Goethe University of Frankfurt am Main

28 May 2014

Used materials:
http://root.cern.ch/
http://ihp-lx.ethz.ch/Stamet/lectureNotes/exercises/rootintro.pdf
http://www-ekp.physik.uni-karlsruhe.de/~jwagner/WS0809/docs/SummerStudents2004.pdf

What is ROOT?

ROOT is an object oriented framework for data analysis
- Read data from different sources
- Write data (persistent object)
- Select data with some criteria
- Produce results as plots, fits, etc...

Support “interactive” (C/C++ , Python) as well as “compiled” usage (C++)

ROOT integrates several tools:
- Random number generators
- Fit methods (Minuit)
- Neural Network framework (TMVA)
- Parallel processing framework (PROOF)
- Vc

Developed and supported by High Energy Physics community
- Homepage with documentation and tutorials: root.cern.ch

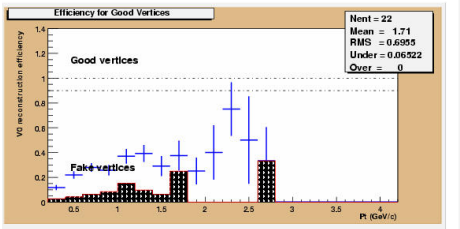
The Site

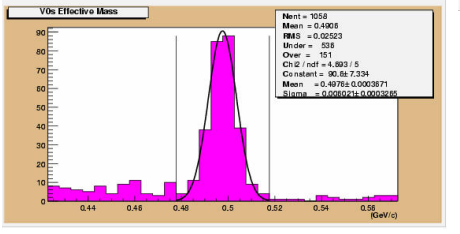
root.cern.ch

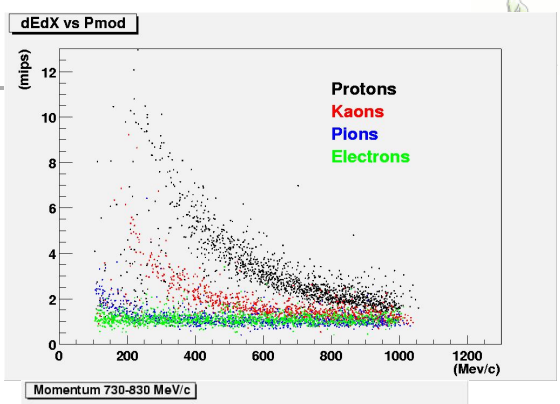
- Latest Version
- Tutorials
- User's Guide
- Reference Guide
- Forum

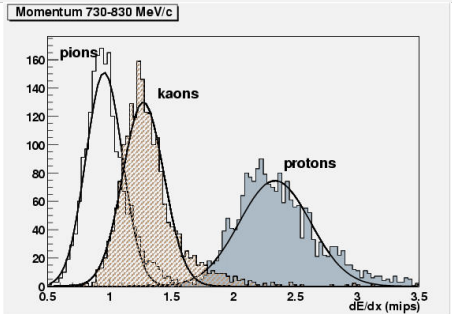
Histograms & Functions

A Data Analysis & Visualisation tool



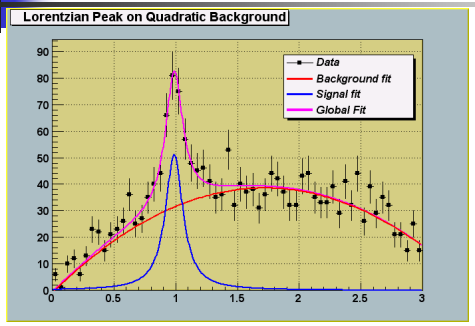


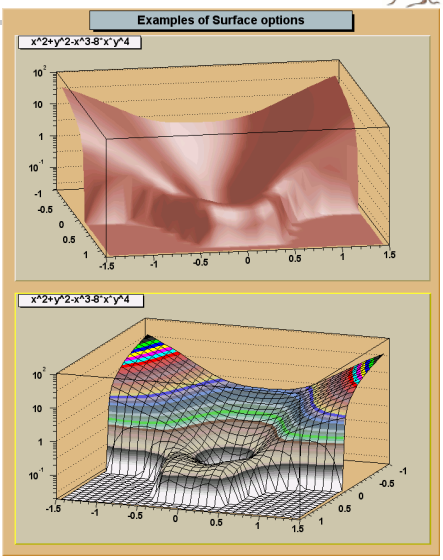


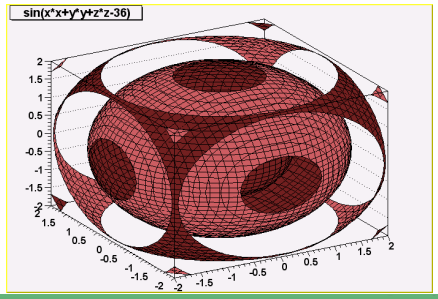


Histograms & Functions

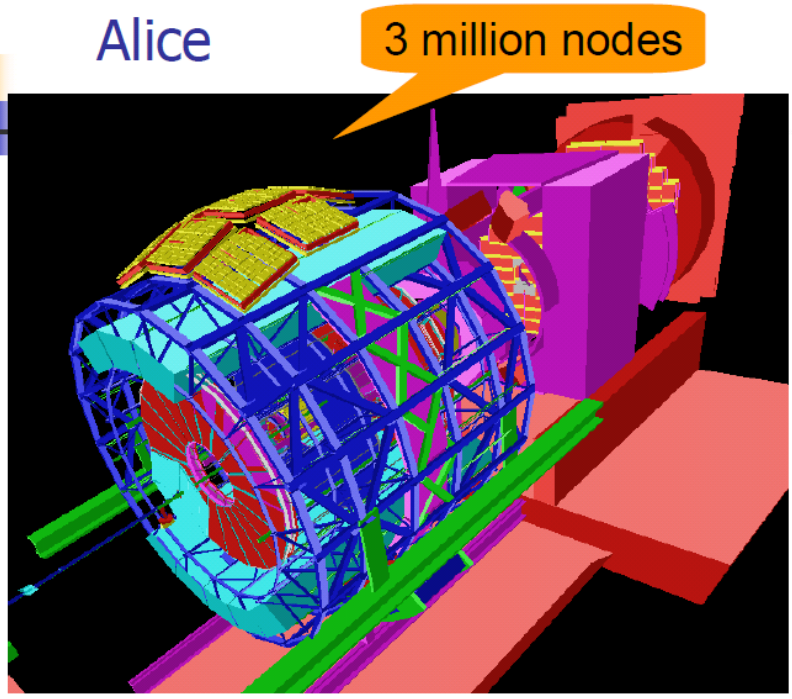
Graphics : 1,2,3-D functions





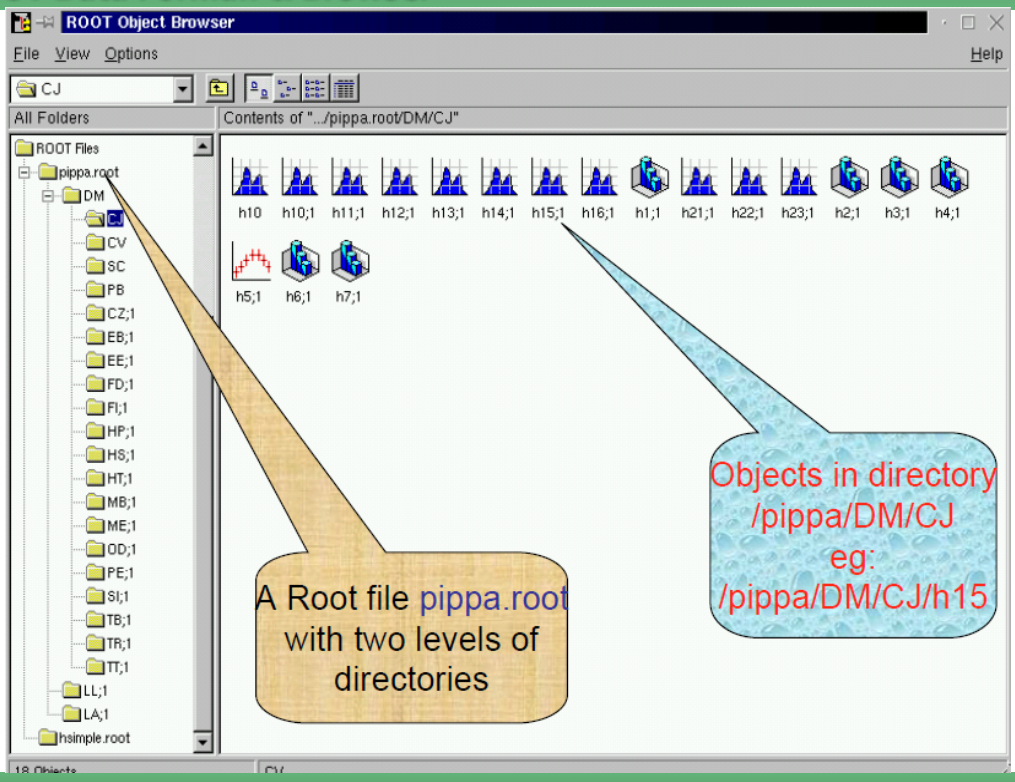


Event Display



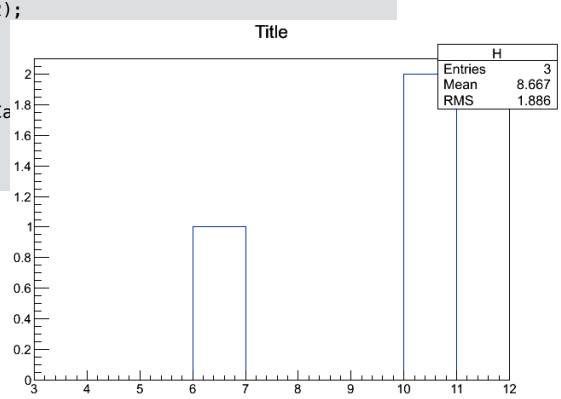
Introduction to ROOT 15

ROOT Data Forman & Browser



ROOT Interactive Console

```
[08:16:13]~$ root -l
root [0] 2 + 2
(const int)4
root [1] sqrt( sin( log( 1.5 ) ) )
(const double)6.28049520113152182e-01
root [2] int s = 0;
root [3] for( int i = 0; i < 10; i++ ) s += i;
root [4] s
(int)45
root [5] TH1F *histo = new TH1F(

TH1F TH1F()
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup)
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, const Float_t* xbins)
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, const Double_t* xbins)
TH1F TH1F(const TVectorF& v)
TH1F TH1F(const TH1F& h1f)
root [5] TH1F *histo = new TH1F("H","Title", 9, 3, 12);
root [6] histo->Fill(10);
root [7] histo->Fill(6);
root [8] histo->Fill(10);
root [9] histo->Draw();
Info in <TCanvas::MakeDefCanvas>:  created default TCa
root [10] .q
[08:17:54]~$
```



CINT is the C++ interpreter of ROOT.
Aclic is the C++ compiler invoked by ROOT.

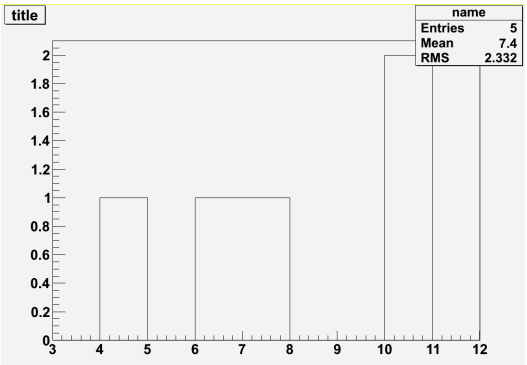
ROOT Compiler

```
[16:23:28]~/Praktikum/Our/Exersise/Exercises/3_ROOT$ root -l –q FittingDemoSimple_0.C+
root [0]
[ compile ]
Processing FittingDemoSimple_0.C+...
Info in <TUnixSystem::ACLiC>: creating shared library /u/ikulakov/Praktikum/Our/Exersise/Exercises/
2_ROOT/./FittingDemoSimple_0_C.so
In file included from /u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/
FittingDemoSimple_0_C_ACLiC_dict.h:34,
                 from /u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/
FittingDemoSimple_0_C_ACLiC_dict.cxx:17:
/u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/./FittingDemoSimple_0.C: In function 'void
FittingDemoSimple_0()':
/u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/./FittingDemoSimple_0.C:25: warning: unused variable
'nBins'
/u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/./FittingDemoSimple_0.C:27: warning: unused variable
'c1'
/u/ikulakov/Praktikum/Our/Exersise/Exercises/2_ROOT/./FittingDemoSimple_0.C:35: warning: unused variable
`fitFcn`

[ run ]
 FCN=93.5605 FROM MIGRAD    STATUS=CONVERGED      54 CALLS          55 TOTAL
                     EDM=1.47795e-17    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0          -3.32978e+00   8.53859e-01   1.65270e-03   7.48073e-09
   2  p1           3.60739e+01   1.71118e+00   7.72138e-04  -2.53062e-09
   3  p2          -1.18800e+01   5.35181e-01   2.79446e-04  -1.69088e-08
```


Histograms

- **TH1**: base classes for 1-D histograms
  - **TH1C**: char based histogram (max content 255)
  - **TH1S**: short based histogram (max content 65 635)
  - **TH1I**: int based histogram (max content 2 147 483 647)
  - **TH1F**: float based histogram (precision 7 digits)
  - **TH1D**: double based histogram (precision 14 digits)
- **TH2**, **TH3** - 2-D and 3-D histograms
- **TProfile**, **TProfile2D** - profile histograms

```
TH1F *hist = new TH1F("name", "title", 9, 3, 12);

hist->Fill(10);
hist->Fill(6);
hist->Fill(7);
hist->Fill(10);
hist->Fill(4);

hist->Draw();
```

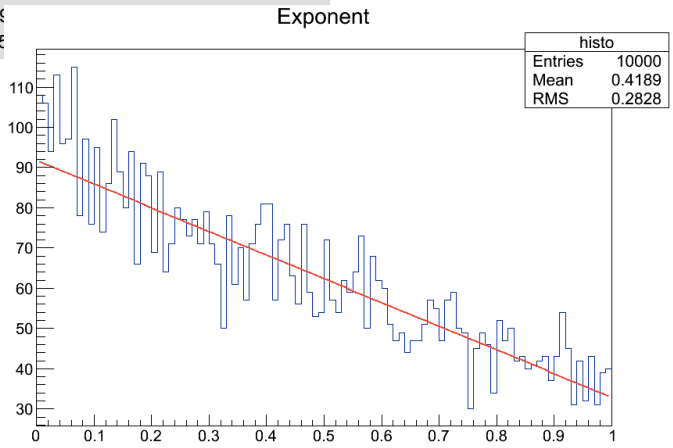


Fitting

```
[08:44:18]~$ root -l
root [0] TH1F *histo = new TH1F("histo", "Exponent",100,0,1);
root [1] TRandom3 rand;
root [2] for(int i=0; i < 10000; i++) histo->Fill( rand.Exp(1) );
root [3] TF1 *fitFcn = new TF1("lineFunction", "[0]*x+[1]" , 0, 1);
root [4] histo->Fit("lineFunction");
Info in <TCanvas::MakeDefCanvas>:  created default T                me c1
 FCN=121.768 FROM MIGRAD    STATUS=CONVERG      .S        32 TOTAL
                EDM=3.4953e-20    STRATEGY= 1     EF              ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           -5.88320e+01   2.69418e+00   6.3570
   2  p1            9.17483e+01   1.73963e+00   4.1047
```

parameters to find

argument

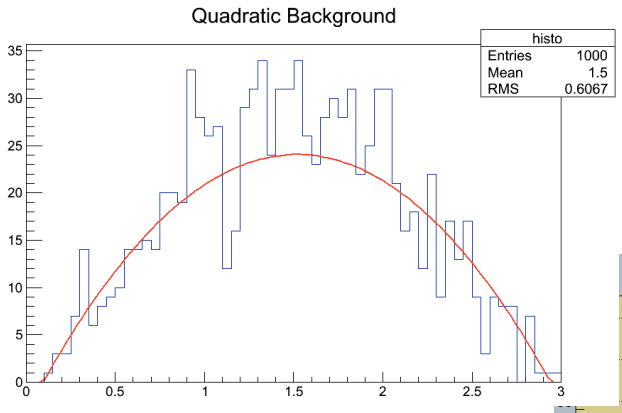


Exercises

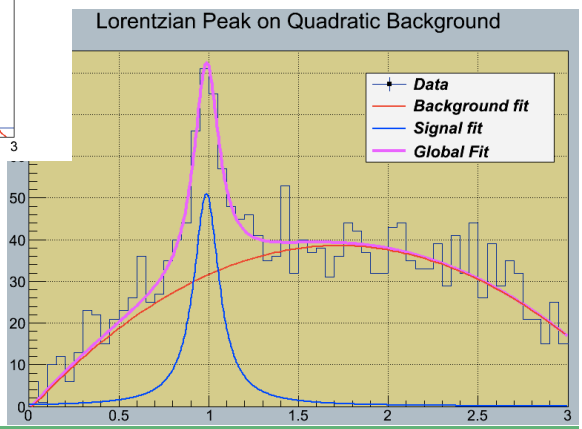
Exersices/4_ROOT

root -l FittingDemoSimple_0.C

FittingDemoSimple_0.C



FittingDemo_1.C



2.4. ROOT Framework

Exercises are located at Exercises/4_ROOT/
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

ROOT Introduction

ROOT is an object-oriented framework for data analysis.[1] ROOT includes abundant functionality for data processing: input, output, plots, curve fitting, minimisation, random generators, matrix computations, MIMD and SIMD.

This manual covers very basic usage of histogram plots and curve fitting and random generators.

ROOT is interpreter based environment, this means that ROOT reads and executes statements one by one. For comparison, compiler read full file, check for semantic errors first, create and executable, which can be executed afterwards separately. ROOT interpreter is called CINT, it is C++ based, so basically all standard C++ statements are allowed in CINT.

ROOT can be used in two different ways:

1. Start ROOT environment at system prompt (terminal):

root -l

then ROOT prompt (you will see “root[0]”) appears and you can input and execute statements from keyboard in the same way as with system prompt. To exit ROOT prompt use **.q** command.

2. Create a ROOT macros and process it with ROOT using:

root -l RootMacrosName.C

then ROOT will look for a function in the macros, which name is same as file name “RootMacrosName()”, and all statements in the function will be executed with interpreter, then ROOT will stop, allowing you to enter additional commands or to exit with .q command.

4_ROOT/FittingDemoSimple_0: description

This exercise show you basic usage of histograms. Histogram is a plot, which shows how many data of each type you have, it is called distribution. For example, it can shows number of days in 12 months, or number of students with each possible heights in cm. In the last case, since height is continuous scale (each student has it’s own height if we measured it precise enough) so called binning should be chosen. In the example above we chose bin size equal to 1 cm and put all students with same height within 1 cm to the one bin, that allows us to calculate height distribution. Bins with size of 10, 3 and 3.3 can be chosen too, for example.

The given macros **FittingDemoSimple_0.cpp** includes required parts of ROOT, describes a function to fit histogram **fitFunction** with and the main function **FittingDemoSimple_0**.

Instead of usual types like **int, float, double**, which can have different precision on different machines ROOT introduces machine independent types: **Int_t, Float_t, Double_t**.

To use graphics with ROOT one needs to declare **TCanvas** object, which has the following constructor:

TCanvas(const char* name, const char* title, Int_t wtopx, Int_t wtopy, Int_t ww, Int_t wh)

here **wtopx, wtopy** are the pixel coordinates of the top left corner of the canvas, **ww** is the canvas size in pixels along X, **wh** is the canvas size in pixels along Y.

Canvas should be allocated dynamically variable, otherwise it would be deleted upon function finish and will not be seen.

---

[1] http://root.cern.ch

The histogram object is called **TH1F**, where H states for histogram, 1 for 1-dimensional, F - for **float** type:

TH1F(const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup)

here **nbins** is number of bins, **xlow** is low edge of first bin, **xup** is upper edge of last bin.
To generate distribution random generator is used **TRandom3** class provides the most optimal random generator in ROOT, its member function **Uniform** provides floating point numbers uniformly distributed from 0 to 1. The generated numbers is added one by one to histogram using **Fill** function.
The histogram is fitted with quadratic function. For this TF1 object is created:

TF1(const char* name, void* fcn, Double_t xmin, Double_t xmax, Int_t npar)

here **fcn** is C++ function name, **xmim**, **xmas** is range of the function argument, **npar** is number of parameters to be fitted. The **fcn** must have two arguments: array of arguments and array of parameters. C++ function **fitFunction** is given to the object here, it has one argument and 3 parameters. The fitting procedure is executed by call of **Fit** function on histogram variable.

Part of the source code of FittingDemoSimple_0.cpp

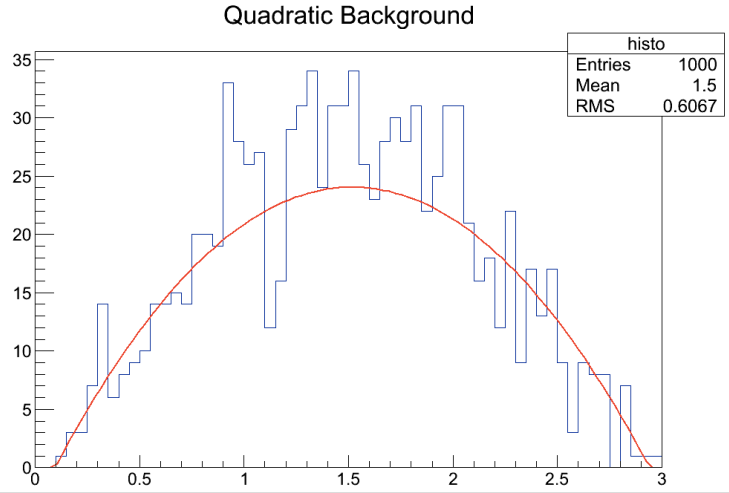
```
9  #include "TH1.h"
10 #include "TMath.h"
11 #include "TF1.h"
12 #include "TLegend.h"
13 #include "TCanvas.h"
14 #include "TRandom3.h"
15
16   // Quadratic function
17   // x   - array of arguments
18   // par - array of parameters
19 Double_t fitFunction(Double_t *x, Double_t *par) {
20    return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
21 }
22
23 void FittingDemoSimple_0() {
24
25    const int nBins = 60;
26
27    TCanvas *c1 = new TCanvas("c1","Fitting Demo",10,10,700,500);
28
29    TH1F *histo = new TH1F("histo", "Quadratic Background",60,0,3);
30
31    TRandom3 rand;
32    for(int i=0; i < 1000; i++) histo->Fill( (rand.Uniform()+rand.Uniform())*1.5
   ); // fill with triangular distribution
33
34      // create a TF1 with the range from 0 to 3 and 3 parameters
35    TF1 *fitFcn = new TF1("fitFcn",fitFunction,0,3,3);
36
37      // fit histo by fitFcn
38    histo->Fit("fitFcn");
39 }
```

Typical output

```
Processing FittingDemoSimple_0.C...
```

Typical output

```
FCN=93.5605 FROM MIGRAD    STATUS=CONVERGED      54 CALLS          55 TOTAL
                     EDM=1.47795e-17    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  p0           -3.32978e+00   8.53859e-01   1.65270e-03   7.48073e-09
  2  p1            3.60739e+01   1.71118e+00   7.72138e-04  -2.53062e-09
  3  p2           -1.18800e+01   5.35181e-01   2.79446e-04  -1.69088e-08
```



4_ROOT/FittingDemo_1: description

This exercise show you more advanced usage canvas options.

Here we use data from a physics experiment, it is saved as a constant array (line 35). The given data contains measurement, which defined by two processes: background process and signal process. Background process gives us quadratic distribution, signal - gives Lorenz distribution. Therefore the data should be fitted by sum of this functions (line 27). Since the function has 6 parameters fitting procedure has no idea about, to help fitting procedure it is better to initialise them by some values, **SetParameters** and **SetParameter** member-functions is used for this. Different options are used for fitting: **0** to do not plot fit immediately, **V** for verbose mode, **+** to save previous fit results, , all list of options and they description can be found at ROOT manual[2]. Creating in addition **background** and **lorentzianPeak** functions we can draw all three of them, showing background and signal components separately (lines 82-89).

The macro also shows how to change colours of canvas (lines 42-44) and style of histogram (lines 48-50), and fitting curves (lines 56-58,76,78,79).

In addition legend is drawn, it contains short description of each object on the picture.

Part of the source code of FittingDemoSimple_0.cpp

```
12 // Quadratic background function
13 Double_t background(Double_t *x, Double_t *par) {
…
18 // Lorenzian Peak function
19 Double_t lorentzianPeak(Double_t *x, Double_t *par) {
…
25 // Sum of background and peak function
26 Double_t fitFunction(Double_t *x, Double_t *par) {
```

[2] http://root.cern.ch/root/html534/TH1.html#TH1:Fit

Part of the source code of FittingDemoSimple_0.cpp

```
27    return background(x,par) + lorentzianPeak(x,&par[3]);
28  }
29
30  void FittingDemo_1() {
31   //Bevington Exercise by Peter Malzacher, modified by Rene Brun
32
33     const int nBins = 60;
34
35     Double_t data[nBins] = { 6, 1,10,12, 6,13,23,22,15,21,
36                              23,26,36,25,27,35,40,44,66,81,
37                              75,57,48,45,46,41,35,36,53,32,
38                              40,37,38,31,36,44,42,37,32,32,
39                              43,44,35,33,33,39,29,41,32,44,
40                              26,39,29,35,32,21,21,15,25,15};
41     TCanvas *c1 = new TCanvas("c1","Fitting Demo",10,10,700,500);
42     c1->SetFillColor(33);
43     c1->SetFrameFillColor(41);
44     c1->SetGrid();
45
46     TH1F *histo = new TH1F("histo",
47        "Lorentzian Peak on Quadratic Background",60,0,3);
48     histo->SetMarkerStyle(21);
49     histo->SetMarkerSize(0.8);
50     histo->SetStats(0);
51
52     for(int i=0; i < nBins;  i++) histo->SetBinContent(i+1,data[i]);
53
54     // create a TF1 with the range from 0 to 3 and 6 parameters
55     TF1 *fitFcn = new TF1("fitFcn",fitFunction,0,3,6);
56     fitFcn->SetNpx(500);
57     fitFcn->SetLineWidth(4);
58     fitFcn->SetLineColor(kMagenta);
…
65     fitFcn->SetParameters(1,1,1,1,1,1);
66     histo->Fit("fitFcn","0");
67
68     // second try: set start values for some parameters
69     fitFcn->SetParameter(4,0.2); // width
70     fitFcn->SetParameter(5,1);   // peak
71
72     histo->Fit("fitFcn","V+");
73
74     // improve the picture:
75     TF1 *backFcn = new TF1("backFcn",background,0,3,3);
76     backFcn->SetLineColor(kRed);
77     TF1 *signalFcn = new TF1("signalFcn",lorentzianPeak,0,3,3);
78     signalFcn->SetLineColor(kBlue);
79     signalFcn->SetNpx(500);
80     Double_t par[6];
```

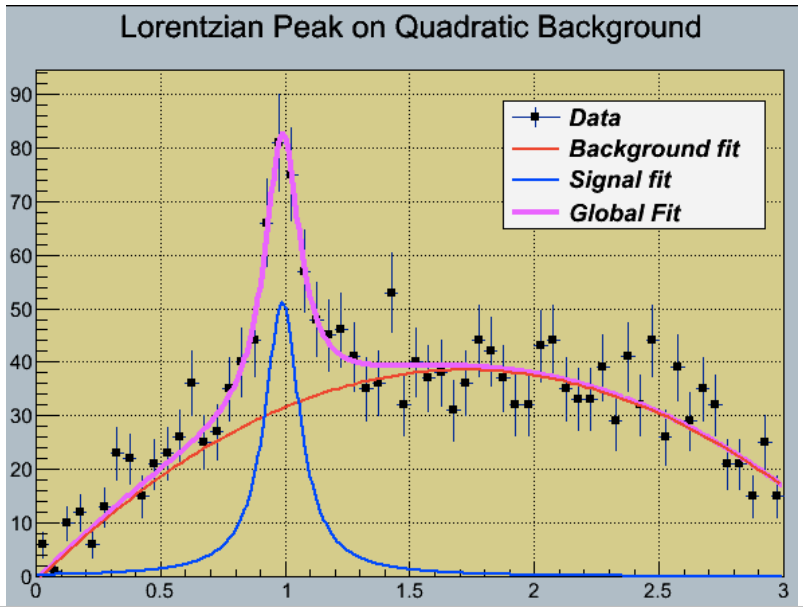
Part of the source code of FittingDemoSimple_0.cpp

```
81
82     // writes the fit results into the par array
83     fitFcn->GetParameters(par);
84
85     backFcn->SetParameters(par);
86     backFcn->Draw("same");
87
88     signalFcn->SetParameters(&par[3]);
89     signalFcn->Draw("same");
90
91     // draw the legend
92     TLegend *legend=new TLegend(0.6,0.65,0.88,0.85);
93     legend->SetTextFont(72);
94     legend->SetTextSize(0.04);
95     legend->AddEntry(histo,"Data","lpe");
96     legend->AddEntry(backFcn,"Background fit","l");
97     legend->AddEntry(signalFcn,"Signal fit","l");
98     legend->AddEntry(fitFcn,"Global Fit","l");
99     legend->Draw();
100
101 }
```

Begin of typical output

```
Processing FittingDemo_1.C...
 FCN=58.9284 FROM MIGRAD    STATUS=CONVERGED     618 CALLS         619 TOTAL
                     EDM=1.54329e-09    STRATEGY= 1  ERROR MATRIX UNCERTAINTY
1.2 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           -8.64715e-01   8.87889e-01   3.02210e-05  -3.15277e-06
```

…



HPC Practical Course
Part 3.1

Open Multi-Processing (OpenMP)

V. Akishina, I.Kisel,
I. Kulakov, M. Zyzak

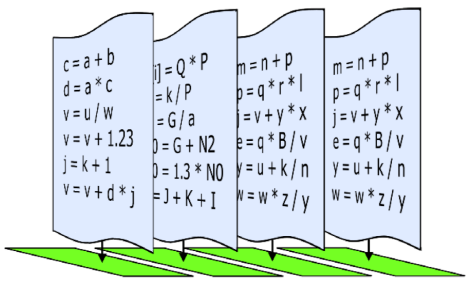
Goethe University of Frankfurt am Main

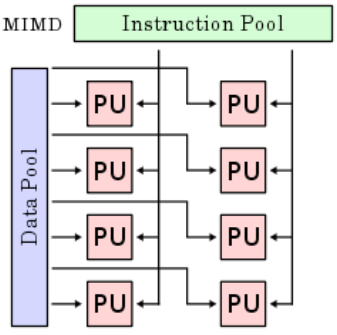
2 Feb 2014

Task Parallelism

Parallelization between cores → Task Parallelism, Parallelization, MIMD

Tools:
- OpenMP
- OpenCL
- Intel Threading Building Blocks (ITBB)
- Pthreads
- Intel Cilk
- Intel Array Building Blocks
- MPI





Performance Characterization

- Performance increasing is characterized with speedup factor
- In ideal case – perfect linear speedup
- Super-linear speedup (usually cache effect)
- Speedup in presence of the serial part
- Number of CPUs is infinite (P → ∞) – the maximal speedup (Amdahl's law)

$$S(P) = \frac{Time(1)}{Time(P)}$$

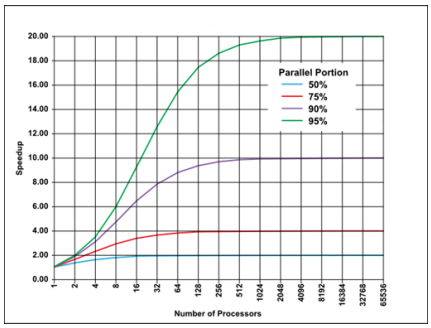
$$S(P) = P$$

$$S(P) > P$$

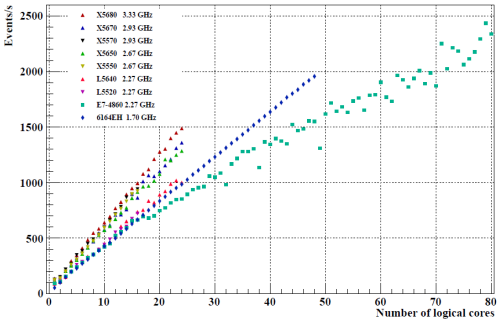
$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

$$S(P) = \frac{1}{\alpha}$$

Amdahl's law



Real-life example – scalability of the CBM track finder on different platforms



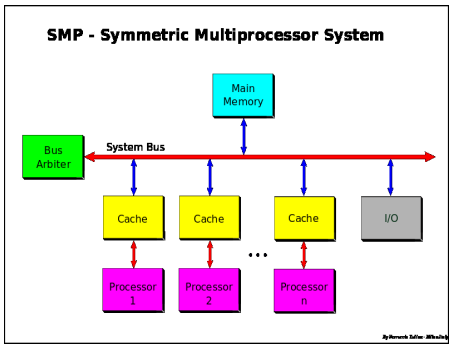
CPU systems

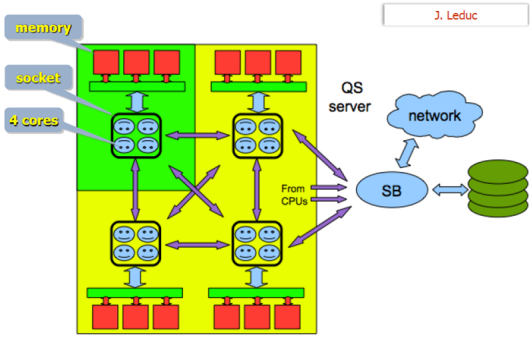
SMP (symmetric multiprocessor) systems:

- Homogeneous
- "Equal-time" access for each processor to the any part of the memory

NUMA (non uniform memory access) systems:

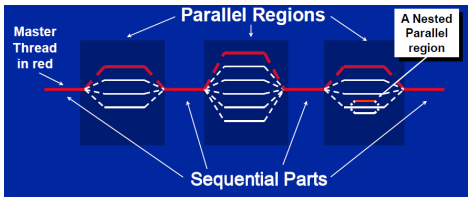
- Heterogeneous
- Non uniform access to different parts of the main memory – different speed, data should be close to the CPU





OpenMP

- API for multi-processing programming
- Supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran
- Supports most processor architectures and operating systems, including Linux, Unix, AIX, Solaris, OS X, and Microsoft Windows platforms
- Most of the constructs in OpenMP are compiler directives:
#pragma omp <construct> [clause [clause]…]
- Function prototypes and types are available only including 1 file:
#include <omp.h>
- Threads communicate by sharing variables
- Programming model – Fork-Join parallelism:



- Compilation flag for g++ – -fopenmp

Simple Example with OpenMP

Without clause. Using runtime function. The number of threads will be set to 4 for all parallel regions in the program.

```
#include <omp.h>
#include <iostream>
using namespace std;

int main ()
{
  omp_set_num_threads(4);

  #pragma omp parallel
  {
    cout << "Hello World" << endl;
  }
}
```

Runtime library routine

Construct "parallel"

With clause. The number of threads will be set to 4 only for one particular parallel region.

```
#include <omp.h>
#include <iostream>
using namespace std;

int main ()
{

  #pragma omp parallel num_threads(4)
  {
    cout << "Hello World" << endl;
  }
}
```

Clause

Worksharing. Loop Construct

- The loop worksharing construct splits up loop iterations among the threads in a team
- The command is #pragma omp for
- Motivation:

Without worksharing:

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart= id * N / Nthrds;
  iend= (id+1) * N / Nthrds;
  if (id == Nthrds-1)
    iend= N;
  for(i=istart;I<iend;i++) { a[i] = a[i] + b[i];}
}
```

With worksharing:

```
#pragma omp parallel
#pragma omp for
   for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

Or:

```
#pragma omp parallel for
   for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

- Using for construct make sure, that iterations are independent:

```
int i, j, A[N];
j = 5;
for (i=0; i<N; i++) {
  j +=2;
  A[i] = func(j);
}
```

→

```
int i, A[N];
#pragma omp parallel for
for (i=0; i<N; i++) {
  int j = 5 + 2*(i+1);
  A[i] = func(j);
}
```


Reduction

- Reduction clause:

reduction(op:list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" .
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- Example:

```
double ave=0.0, A[N];
int i;
#pragma omp parallel for reduction (+:ave)
for (i=0; i<N; i++) {
  ave + = A[i];
}
ave = ave/N;
```

Reduction operands	
Operator	Initial value
+	0
*	1
-	0
&	0
|	0
^	0
&&	1
||	0

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data.
- High level synchronization:
  - critical – only one thread at a time can enter a critical region;
  - atomic – only one thread can operate with a memory location marked by atomic;
  - barrier – each thread waits until all threads arrive;
  - ordered – executes in the sequential order.
- Low level synchronization:
  - Flush – forces data to be updated in memory so other threads see the most recent value;
  - locks (both simple and nested).

High Level Synchronization. Examples

Critical. Only 1 thread at a time calls f().

```
#pragma omp parallel
{
  … some code …
  for(int i=0; i<N; i++) {
    … some code …
    #pragma omp critical
      f(…);
  }
}
```

Atomic. Protects only read/update of X.

```
#pragma omp parallel
{
  double tmp;
  … some code …
  #pragma omp atomic
    X+= tmp;
}
```

Barrier.

```
#pragma omp parallel
{
  … some code …
  #pragma omp barrier
  … some code …

  #pragma omp for
  for(int i=0; i<N; i++) { … some code … }

 #pragma omp nowait
  #pragma omp for
  for(int i=0; i<N; i++) { … some code … }
}
```

All tasks will wait here

Implicit barrier

No implicit barrier due to nowait

Implicit barrier

Synchronization. Locks

- Simple lock runtime library routines: omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks: omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()
- Example on the simple lock:

```
#include <omp.h>
#include <iostream>
using namespace std;

int main ()
{
  omp_set_num_threads(4);

  omp_lock_t lck;
  omp_init_lock(&lck);
  #pragma omp parallel
  {
    omp_set_lock(&lck);
      cout << "Hello World" << endl;
    omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
}
```


Data Sharing Clauses

- shared – the variable is shared between threads, can not be applied to omp for loop.
- private – creates a new copy of a variable for each thread, value is uninitialized:

```
void f() {
  int tmp = 0;
  #pragma omp for private(tmp)
  for (int j = 0; j < 1000; ++j)
    tmp += j;   <-- Is not initialized!
  printf("%d\n", tmp);
}
```

- firstprivate – like private, but initialized with the value of the master thread:

```
void f() {
  int tmp = 0;
  #pragma omp for firstprivate(tmp)
  for (int j = 0; j < 1000; ++j)
    tmp += j;   <-- Here initialized with 0
  printf("%d\n", tmp);
}
```

- lastprivate – like private, but the last value will be saved to a global variable.
- default (private or none) – sets the default clause for all variables

Loop Worksharing Constructs: The Schedule Clause

The schedule clause affects how loop iterations are mapped onto threads:

- schedule(static [,chunk])
  - Deal-out blocks of iterations of size "chunk" to each thread.
- schedule(dynamic[,chunk])
  - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
- schedule(guided[,chunk])
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
- schedule(runtime)
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0)

Common Runtime Library Routines

- Runtime environment routines:
  - Modify/Check the number of threads:

  omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - shows are we in an active parallel region:

  omp_in_parallel()
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?

  omp_set_dynamic, omp_get_dynamic();
  - Shows the number of processors in the system:

  omp_get_num_procs()

- More of them you can find here: https://computing.llnl.gov/tutorials/openMP/#RunTimeLibrary

CPU affinity

- To avoid jumping of threads from one core to another we should set a CPU affinity.
- For this we can use, for example, pthreads:

```
int cpuId = …
pthread_t thread  = pthread_self(); // get the current thread
cpu_set_t cpuset; // declare a cpu_set_t variable
CPU_ZERO(&cpuset); //macros initializes this variable
CPU_SET(cpuId, &cpuset); //set the cpuset variable according to the cpuId
int s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset); //set the affinity
```


Exercise 0. Hello World

```
#include <iostream>
using namespace std;

int main() {

  int id = 0;
  cout << " Hello world " << id << endl;

  return 0;
}
```

- Parallelize the program with OpenMP, create 10 threads.
- Synchronize threads using omp critical. Compare the results with and without synchronization.
- Get the id of a current thread and print it out.

Exercise 1. Bugs

- Find bugs and fix them

Exercise 2.

- The program calculates an integral:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- Tasks:
  - parallelize with OpenMP using only #pragma omp parallel;
  - parallelize with OpenMP making as small changes in the program as possible.

Exercise 3. Matrix

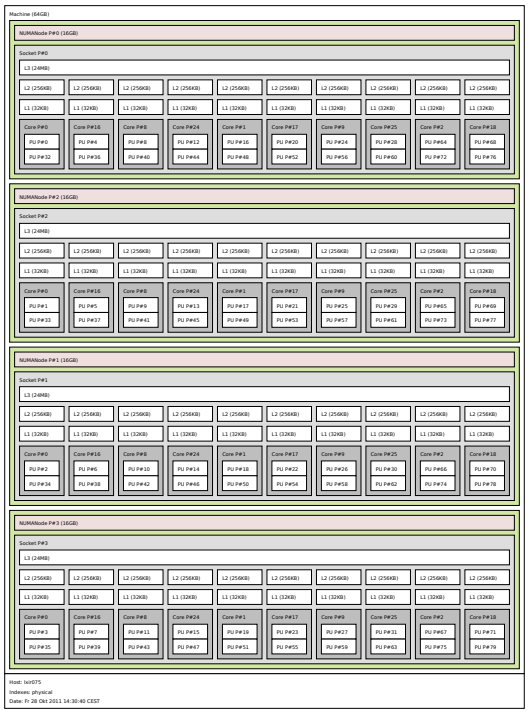
- Parallelize the program Matrix, which we already SIMDized in previous exercises (Exercise/Exercises/3_Vc/1_Matrix)
- Compare the time of the single-core scalar program and the fully parallelized and vectorized program

Exercise 4. CBM KF

- Parallelize the already SIMDized program (we have done it last week) using OpenMP between cores.
- Compare the speed with a single-core program.
- Connect to the server with 80 cores (ask Maksym).
- Go to the folder /u/mzyzak/HPC/[i]
- Configure the environment: . AddVcRoot.sh
- Go to the folder hltsse/TimeHisto
- Run the bash script, which measures the scalability of the program:

  . make_data_omp.sh
- Check the scalability: root –l make_timehisto_stat_complex.C
- Set CPU affinity without specifying scheduler. Check the scalability.
- Set dynamic scheduler without specifying the chunk size, with chunk size 10, 100 and 1000. Check the scalability.
- Set guided scheduler without specifying the chunk size (and with chunk of size 10 and 100).
- Set static scheduler without specifying the chunk size (and with chunk of size 10).
- Change the mode of speed measurement: each thread will measure its own speed, then the speed will be added. For this uncomment the line 2 in Fit.cxx. Remeasure the scalability without CPU affinity and without specifying a scheduler.
- Remeasure scalability with the dynamic scheduler (without chunk size, with chunk of size 10).

Topology of the Server



3.1. OpenMP

Exercises are located at Exercises/5_OMP/
Solutions are located at Exercises/5_OMP/***/***_solution.cpp
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

OpenMP Introduction

OpenMP is an API, which consist of a set of compiler directives, library routines and environment variables that are used to create multithreaded applications on multiprocessor systems with a shared memory. It defines a simple interface that allows to parallelise tasks between cores of the CPU.

The OpenMP framework supports most processor architectures and operating systems, including Linux, Unix, AIX, Solaris, OS X, and Microsoft Windows platforms.

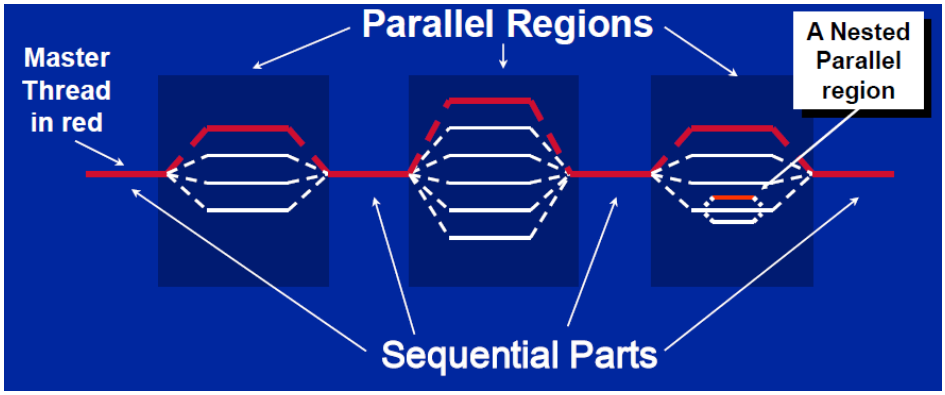


Fig. 1. Fork-join parallelism model of OpenMP.

The programming model of OpenMP is the fork-join parallelism: the master thread is created, when a program starts and when the calculation should be parallelised additional threads are created and the task is distributed between them (see Fig. 1). The nested parallelism is also possible within the OpenMP programming model. The parts of the code, which should be executed in parallel are marked with the directives of the compiler preprocessor:

#pragma omp parallel.

Threads in the OpenMP programming model are communicated by sharing variables. Each variable can be private or shared. The shared variables are seen by all threads. If the variable is declared as private a local copy of it is created by each thread.

Most of the constructs in OpenMP are compiler directives:

#pragma omp <construct> [clause [clause]...].

The implementation of programs using OpenMP constructs allow to keep the code unchangeable with respect to the single core version. In this case only an appropriate flag should be added during the compilation to enable the OpenMP directives. OpenMP includes such constructs, for example, as **omp for**, **omp reduction(op:list), omp num_threads(), omp scheduler [clause]**, etc. These directives allow to control the parallel regions, to collect the data from different threads, to synchronise threads, setting of the scheduler.

omp for construct provides a simple and easy to use interface for loop parallelisation. Let us compare two approaches of parallelisation - with worksharing construct and without:

Without worksharing:	With worksharing:

#pragma omp parallel
{
int id, i, Nthrds, istart, iend;
id = omp_get_thread_num();
Nthrds = omp_get_num_threads();
istart= id * N / Nthrds;
iend= (id+1) * N / Nthrds;
if (id == Nthrds-1)
iend= N;
for(i=istart;I<iend;i++) { a[i] = a[i] + b[i];}
}
	#pragma omp parallel
	#pragma omp for
	for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
	
	Or:
	
	#pragma omp parallel for
	for(i=0;i<N;i++) { a[i] = a[i] + b[i];}

Also it is important to keep iteration completely independent using **omp for**. Otherwise the result will be incorrect.

omp num_threads() clause set the number of threads for the current parallel region.

Using OpenMP clauses it is possible to define if the variable shared or private:

- **shared(list)** - the variable is shared between threads, can not be applied to omp for loop;
- **private(list)** - creates a new copy of a variable for each thread, value is uninitialised;
- **firstprivate(list)** - like private, but copies are initialised with the value of the master thread;
- **lastprivate(list)** - like private, but the last value will be saved to a copy of the variable in the master thread;
- **default (private or none)** - sets the default clause for all variables;

By default all variables are shared.

omp reduction(op:list) specifies the list of variables, that should be reduced at the end of the parallel region. Each thread creates a local copy of the variable, when the parallel region is finished the master thread collects values stored in the local copies into it's variable, which is initialised according to the table (depending on the specified operation):

Reduction operands	
Operator	Initial value
+	0
*	1
–	0
&	0
|	0
^	0
&&	1
||	0

omp schedule [clause] defines the way, how the data is distributed between threads:

- **schedule(static [,chunk])** - blocks of iterations of size "chunk" are gradually distributed to each thread;
- **schedule(dynamic[,chunk])** - each thread grabs "chunk" iterations off a queue, when it finished the previous work, until all iterations have been handled;
- **schedule(guided[,chunk])** - threads dynamically grab blocks of iterations, the size of the block starts large and shrinks down to size "chunk" as the calculation proceeds;
- **schedule(runtime)** - schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).

In order to control the race conditions the framework contains a set of tools for a thread synchronisation including high level directives, like **omp critical, omp atomic, omp barrier**, and a set of simple lock functionality for a low level control:

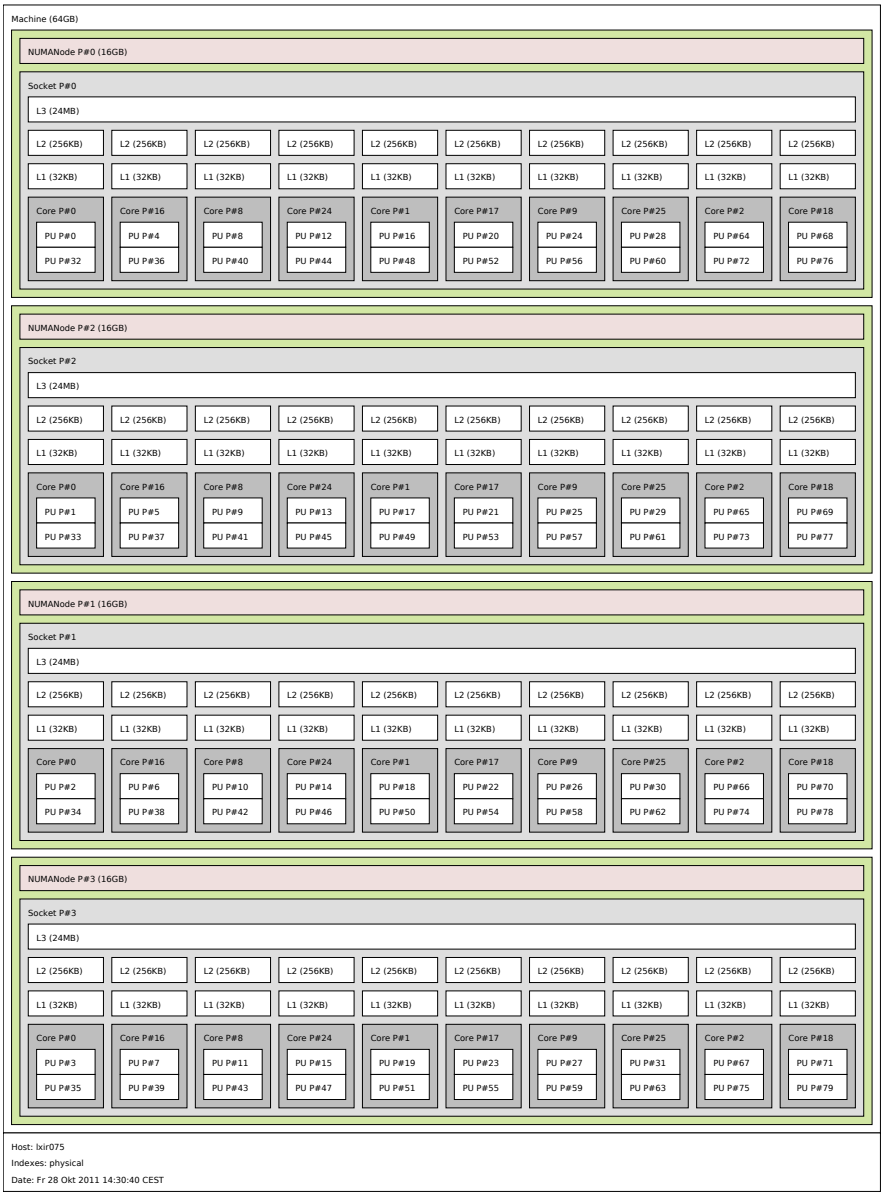


Fig. 2. The structure of the lxir075 server.

- **critical** - only one thread at a time can enter a critical region;
- **atomic** - only one thread can operate with a memory location marked by atomic;
- **barrier** - each thread waits at the barrier until all threads arrive;
- **ordered** - threads execute in the sequential order;
- **flush** - forces data to be updated in memory so other threads see the most recent value;
- **locks** (both simple and nested) - allows to lock the part of the code, only one thread can enter the locked region; are useful, for example, if some elements of the array shared between threads should be locked.

OpenMP contains also a set of runtime library routines which allow, for instance, setting and to checking the number of threads, checking the maximum number of threads, getting the number of a current thread, checking the number of cores in a computer, to control the nested parallelism, operating with the simple

lock functionality etc. They can be available by including only one file: **#include <omp.h>**. The most useful functions for the practicum are:

omp_set_num_threads() - set the number of thread created in the following parallel regions (if the number of threads is not specified by the corresponding clause there);

omp_get_num_threads() - obtain number of threads running on the current parallel region;

omp_get_thread_num() - returns id of the thread.

More of the functions together with their description can be found here:
https://computing.llnl.gov/tutorials/openMP/#RunTimeLibrary

NUMA architecture

Most of modern many and multi-core servers are heterogeneous systems. An access speed to different regions of memory in such systems is different for the different processing elements (for CPU cores or for CPUs themselves). In order to achieve a maximum speed of computation, it is necessary to control the distribution of memory between tasks. For the control the topology of a server should be known.

The topology of NUMA systems is shown on an example of the lxir075.gsi.de server (GSI, Darmstadt, Germany) on Fig. 2. The server is equipped with four Intel Xeon E7-4860 CPUs (10 physical cores at 2.27 GHz). Each physical core of this CPU can run simultaneously two threads with the hyperthreading technology. Therefore two logical cores correspond to each physical core. Each physical core has 32 kB of the level one (L1) cache memory and 256 kB of the level two (L2) cache memory shared between two logical cores. Each CPU has 24 MB of the level three (L3) cache memory, which is shared between all cores of the CPU. The total RAM of 64 GB is equally distributed between CPUs (16 GB for each of them).

During runtime the operating system can move threads from one logical core to another depending on the load of cores. It can happen, that the thread can jump to another CPU. For the NUMA architectures it is preferable to use only local RAM for the maximum performance. And if the thread would be moved to another CPU all data, which was used by this thread, would be located in the remote RAM. In order to prevent such jumps, which ruin the performance, each thread can be pined to a certain core. OpenMP itself doesn't contain tools for pinning. In order to do so for example the Pthreads library can be used. To use the library the corresponding header file should be included:

#include "pthread.h"

Then the pinning can be done using pthread_setaffinity_np() function:

```
int cpuId = … // index of the logical core
pthread_t thread  = pthread_self(); // get the current thread
cpu_set_t cpuset; // declare a cpu_set_t variable
CPU_ZERO(&cpuset); //macros initialises this variable
CPU_SET(cpuId, &cpuset); //set the cpuset variable according to the cpuId
int s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset); //set the affinity
```

5_OMP/0_HelloWorldParallel: description

The first exercise on OpenMP is aimed to demonstrate how to create the parallel regions and synchronise them. It is proposed to parallelise a simple "Hello word" program:

Part of the source code of HelloWorld.cpp

```
13  #include <iostream>
14  using namespace std;
15
16  int main() {
17
18    int id = 0;
19    cout << " Hello world " << id << endl;
20
21    return 0;
22  }
```

The tasks for the exercise are:

- Parallelise the program with OpenMP, create 10 threads.
- Synchronise threads using **omp critical**. Compare the results with and without synchronisation.
- Get the id of a current thread and print it out. Use variable **id** for this.

5_OMP/0_HelloWorldParallel: solution

The first task is solved by including **omp.h** and adding parallel region:

Part of the source code of HelloWorld_solution1.cpp

```
12 #include <omp.h>
13 #include <iostream>
14 using namespace std;
15
16 int main() {
17
18   #pragma omp parallel num_threads(10)
19   {
20     int id = 0;
21
22     cout << " Hello world " << id <<  endl;
23   }
24
25   return 0;
26 }
```

Such program will print symbols on the screen chaotically, because all 10 threads try to do this at the same time and only one of them can print at the current moment. To prevent such situation the threads should be synchronised. For example, omp critical can be added to the parallel region just before the printing:

Part of the source code of HelloWorld_solution2.cpp

```
16 int main() {
17
18   #pragma omp parallel num_threads(10)
19   {
20     int id = 0;
21
22     #pragma omp critical
23     cout << " Hello world " << id <<  endl;
24   }
25
26   return 0;
27 }
```

To obtain the id of the thread the function **omp_get_thread_num()** should be used. Each thread has it's own id, therefore the function should be called in the parallel region by each thread individually. Then id is stored to the local variable of each thread and printed on the screen:

Part of the source code of HelloWorld_solution3.cpp

```
16 int main() {
17
18   #pragma omp parallel num_threads(10)
19   {
```

Part of the source code of HelloWorld_solution3.cpp

```
20      int id = omp_get_thread_num();
21
22      #pragma omp critical
23      cout << " Hello world " << id <<  endl;
24    }
25
26    return 0;
27  }
```

5_OMP/1_Bugs: description

To illustrate different features of OpenMP and get familiar with it's functionality a set of simple programs containing bugs was created:

bug1.cpp - a program creates two arrays: **input** is filled with random numbers, **output** is filled by copying **input** element-wise in a parallel loop. Because of the bug entries in the **output** differ from the **input** and the program reports, that the output array is not correct.

bug2.cpp - a program fills two arrays: one in a scalar loop (outputScalar), another - in a parallel (outputParallel). Each element of arrays is filled with a sum of indices of all previous elements normalised on the current index. Because of the bug the scalar and parallel results are not the same.

bug3.cpp - a program creates an input array and fills it with random numbers. After this it calculates in a scalar loop the sum over all elements of the array and stores into the outputScalar array corresponding element of the input normalised by the sum. The same calculations are also done in the parallel loop but with a bug and the result is stored in a outputParallel array. Because of the bug the scalar and parallel results are not the same.

bug4.cpp - a program do the same, as the previous, but the **omp for** loop is replaced here with a parallel construct only. Again, due to the bug in a parallel region results calculated in the scalar and parallel loops are not the same.

The task for this exercise is to find and fix the bugs.

5_OMP/1_Bugs: solution

bug1.cpp - in the initial program the number of elements in an array **N** is declared **private** in a **parallel for** construct:

Part of the source code of bug1.cpp

```
33    #pragma omp parallel private(N) num_threads(NThreads)
34    {
35      #pragma omp for
36      for(int i=0; i<N; i++)
37        output[i] = input[i];
38    }
```

With such declaration local variables **N** of the treads will not be initialised. It should be declared as **firstprivate** in order to initialise each local copy with a value of the master thread.

Part of the source code of bug1_solution.cpp

```
33    #pragma omp parallel firstprivate(N) num_threads(NThreads)
34    {
35      #pragma omp for
36      for(int i=0; i<N; i++)
37        output[i] = input[i];
38    }
```

bug2.cpp - the iterations of the loop in the parallel region are not independent:

Part of the source code of bug2.cpp

```
37    #pragma omp parallel num_threads(NThreads)
38    {
39      #pragma omp for
40      for(int i=1; i<N; i++)
41      {
42        tmp += i;
43        outputParallel[i] = float(tmp)/float(i);
44      }
45    }
```

The variable **tmp** in current implementation depends on all previous iteration. Therefore when the loop is divided into several parts in the parallel region, values of local variables **tmp** are corrupted. Taking into account that value of **tmp** is a sum of the arithmetical progression, the iterations can be rewritten in an independent way:

Part of the source code of bug2_solution.cpp

```
37    #pragma omp parallel num_threads(NThreads)
38    {
39      #pragma omp for
40      for(int i=1; i<N; i++)
41      {
42        tmp = (1+i)*i/2;
43        outputParallel[i] = float(tmp)/float(i);
44      }
45    }
```

bug3.cpp - in the initial program in the **omp for** construct during the sum calculation **nowait** clause destroys the implicit barrier and some threads start to fill the outputParallel array when the sum is not yet correctly calculated:

Part of the source code of bug3.cpp

```
46    sum = 0;
47    #pragma omp parallel firstprivate(N) num_threads(NThreads)
48    {
49      #pragma omp for nowait
50      for(int i=0; i<N; i++)
51      {
52        #pragma omp atomic
53        sum += input[i];
54      }
55
56      #pragma omp for
57      for(int i=0; i<N; i++)
58        outputParallel[i] = input[i]/sum;
59    }
```

The solution would be to get rid of the **nowait**. For example, it can be done like this:

Part of the source code of bug3_solution.cpp

```
46    sum = 0;
47    {
48
```

Part of the source code of bug3_solution.cpp

```
49      for(int i=0; i<N; i++)
50      {
51        sum += input[i];
52      }
53
54      #pragma omp parallel firstprivate(N) num_threads(NThreads)
55      #pragma omp for
56      for(int i=0; i<N; i++)
57        outputParallel[i] = input[i]/sum;
58    }
```

bug4.cpp - unlike the previous exercise, here the barrier is missed:

Part of the source code of bug4.cpp

```
57    sum = 0;
58    #pragma omp parallel num_threads(NThreads)
59    {
60      int id, i, Nthrds, istart, iend;
61      id = omp_get_thread_num();
62      Nthrds = omp_get_num_threads();
63      istart= id * N / Nthrds;
64      iend= (id+1) * N / Nthrds;
65      if (id == Nthrds-1)
66        iend= N;
67      float sumLocal = 0;
68
69      CalcuateSum(input,istart, iend, sumLocal);
70
71      #pragma omp atomic
72      sum += sumLocal;
73
74      #pragma omp for
75      for(int i=0; i<N; i++)
76        outputParallel[i] = input[i]/sum;
77    }
```

Again, because of the missed barrier threads will continue without waiting for sum calculation to be completed. To prevent this the barrier should be put after the line 72:

Part of the source code of bug4_solution.cpp

```
57    sum = 0;
58    #pragma omp parallel num_threads(NThreads)
59    {
60      int id, i, Nthrds, istart, iend;
61      id = omp_get_thread_num();
62      Nthrds = omp_get_num_threads();
63      istart= id * N / Nthrds;
64      iend= (id+1) * N / Nthrds;
65      if (id == Nthrds-1)
66        iend= N;
67      float sumLocal = 0;
68
69      CalcuateSum(input,istart, iend, sumLocal);
70
```

Part of the source code of bug4_solution.cpp

```
71      #pragma omp atomic
72      sum += sumLocal;
73
74      #pragma omp barrier
75
76      #pragma omp for
77      for(int i=0; i<N; i++)
78        outputParallel[i] = input[i]/sum;
79    }
```

5_OMP/2_Pi: description

The loops can be parallelised using OpenMP in two ways: manually (using only omp parallel construct and OpenMP functions) and using constructs together with clauses. In order to illustrate the difference it is proposed to parallelise a simple program using these two approaches.

The initial program is a scalar code to calculate the value of π by a formula:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

Part of the source code of pi.cpp

```
19  #include <stdio.h>
20  #include <omp.h>
21
22  static long num_steps = 100000000;
23  double step;
24  int main ()
25  {
26    int i;
27    double x, pi, sum = 0.0;
28    double start_time, run_time;
29
30    step = 1.0/(double) num_steps;
31
32    start_time = omp_get_wtime();
33
34    for (i=1;i<= num_steps; i++){
35      x = (i-0.5)*step;
36      sum = sum + 4.0/(1.0+x*x);
37    }
38
39    pi = step * sum;
40    run_time = omp_get_wtime() - start_time;
41    printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
42  }
```

The tasks for the exercise are:

- parallelise the initial program using only **omp parallel** construct and OpenMP functions;
- parallelise the initial program making as less changes, as possible.

5_OMP/2_Pi: solution

For the solution of the first task it is necessary to define the number of threads (line 30), allocate variables to store the temporary sums calculated by each thread (line 31) and set the number of threads (line 32). In a parallel region it is needed to obtain the id of the thread (line 35, it is defined from zero to (nThreads-1)), calculate manually the maximum index in the loop-range of the current thread (line 37) and then perform the calculations. When calculating x the index i should be modified taking into account the offset for the current thread (line 38). In the end local sums should be summed up together (lines 43 - 44).

Part of the source code of pi_solution1.cpp

```
15 #include <stdio.h>
16 #include <omp.h>
17
18 static long num_steps = 100000000;
19 double step;
20 int main ()
21 {
22
23   double pi, sum = 0.0;
24   double start_time, run_time;
25
26   step = 1.0/(double) num_steps;
27
28   start_time = omp_get_wtime();
29
30   const int nThreads = 2;
31   double sums[nThreads] = {0};
32   omp_set_num_threads(nThreads);
33   #pragma omp parallel
34   {
35     int id = omp_get_thread_num();
36
37     for (int i=1;i<= num_steps/nThreads; i++){
38       double x = (i + id* num_steps/nThreads -0.5)*step;
39       sums[id] = sums[id] + 4.0/(1.0+x*x);
40     }
41   }
42
43   for (int i = 0; i < nThreads; i++ )
44     sum += sums[i];
45
46   pi = step * sum;
47   run_time = omp_get_wtime() - start_time;
48   printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
49 }
```

The solution of the second task would be to add only one line before the loop (line 30):

Part of the source code of pi_solution2.cpp

```
15 #include <stdio.h>
16 #include <omp.h>
17
18 static long num_steps = 100000000;
19 double step;
20 int main ()
```

Part of the source code of pi_solution2.cpp

```
21  {
22    int i;
23    double x, pi, sum = 0.0;
24    double start_time, run_time;
25
26    step = 1.0/(double) num_steps;
27
28    start_time = omp_get_wtime();
29
30  #pragma omp parallel for reduction(+ : sum) private(i,x)
31    for (i=1;i<= num_steps; i++){
32      x = (i-0.5)*step;
33      sum = sum + 4.0/(1.0+x*x);
34    }
35
36    pi = step * sum;
37    run_time = omp_get_wtime() - start_time;
38    printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
39  }
```

In the parallel region x and i should be declared as private and sum should be reduced. It is also worth to notice, that such code will compile and work even when OpenMP is disabled.

5_OMP/3_Matrix: description

This exercise is aimed to demonstrate the full power of CPU: the already vectorised exercise with matrix calculation (similar to 2_Vc/1_Matrix) should be parallelised between cores with OpenMP. In addition to the vectorised code the program contains now the part prepared for parallelisation:

Part of the source code of Matrix.cpp

```
75     ///OpenMP
76    TStopwatch timerOMP;
77    for( int ii = 0; ii < NIter; ii++ ) // repeat several times to improve time
      measurement precision
78    {
79        //TODO modify the code below using OpenMP
80      for( int i = 0; i < N; i++ ) {
81        for( int j = 0; j < N; j+=float_v::Size ) {
82            float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
83            float_v &cVec = (reinterpret_cast<float_v&>(c_omp[i][j]));
84            cVec = sqrt(aVec);
85        }
86      }
87    }
88    timerOMP.Stop();
```

5_OMP/3_Matrix: solution

Since the calculations are absolutely independent it is enough to add only one line to the initial code (line 79):

Part of the source code of Matrix_solution.cpp

```
75     ///OpenMP
```

Part of the source code of Matrix_solution.cpp

```
76    TStopwatch timerOMP;
77    for( int ii = 0; ii < NIter; ii++ ) // repeat several times to improve time
      measurement precision
78    {
79  #pragma omp parallel for num_threads(omp_get_num_procs())
80      for( int i = 0; i < N; i++ ) {
81        for( int j = 0; j < N; j+=float_v::Size ) {
82            float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
83            float_v &cVec = (reinterpret_cast<float_v&>(c_omp[i][j]));
84            cVec = sqrt(aVec);
85        }
86      }
87    }
88    timerOMP.Stop();
```

5_OMP/4_CBM_KF: description

In this exercise we will parallelise between cores already SIMDized KF track fitter (see exercise 3_Vc/5_CBM_KF) using OpenMP. In the ideal case the program should scale linearly with respect to the number of cores in a CPU. For this a speed measurement should be provided. There are two possible running scenarios:

- a program receives a set of data and operates on it for a short amount of time; in this case we are interested on the total speed of the program and the edge effects can not be neglected (they appear close to the finish, for example, if tasks are distributed between threads in an non optimal way, especially on CPUs with hyperthreading);
- a program starts threads and provides an input data to them, the data is updating with a time and threads can work without any pause for several hours or days; in this case we can neglect edge effect, therefor are interested in a speed of each thread and adding them together can estimate the total speed.

The SIMD KF program can measure the speed in both modes. For this a compiler preprocessor macros #define TOTALTIME is added to the Fit.cxx file. For the first case the macros should be commented, for the second mode - uncommented.

The tasks for the exercise are:

1. parallelise between cores the already SIMDized program using OpenMP; in order to do this the data copying should be parallelised:

Part of the source code of Fit.cxx

```
208     //TODO parallelize data copying
209     {
210         for ( int j = 0; j < tasks; ++j ) {
211             for ( int i = 0; i < NCopy; ++i ) {
212                 if ( j * NCopy + i > MaxNTracks ) {
213                     continue;
214                 }
215                 vTracks[j * NCopy + i] = vTracks[i];
216                 vMCTracks[j * NCopy + i] = vMCTracks[i];
217                 NTracks++;
218             }
219         }
220     }
```

and the fitting itself:

Part of the source code of Fit.cxx

```
444   //TODO parallelize calculations
```

Part of the source code of Fit.cxx

```
445   {
446     timerLocal.Start();
447     for( iV=0; iV<NTracksV; iV++ ){ // loop on set of 4 tracks
448       for( ifit=0; ifit<NFits; ifit++){
449          Fit( TracksV[iV], vStations, NStations );
450        }
451        nFittedTracks += vecN;
452     }
453     timerLocal.Stop();
454     double cpuTimeLocal = timerLocal.CpuTime();
455     double realTimeLocal = timerLocal.RealTime();
456
457     if(fabs(cpuTimeLocal)<1.e-8) cpuTimeLocal = realTimeLocal;
458
459 #ifdef OMP
460     int curThredNum = omp_get_thread_num();
461 #else
462     int curThredNum = 0;
463 #endif
464     if(cpuTimeLocal > 0)
465       fitSpeedCPU[curThredNum] = nFittedTracks/cpuTimeLocal;
466     else
467       fitSpeedCPU[curThredNum] = 0;
468
469     if(realTimeLocal > 0)
470       fitSpeedReal[curThredNum] = nFittedTracks/realTimeLocal;
471     else
472       fitSpeedReal[curThredNum] = 0;
473   }
```

2. set the CPU affinity without specifying scheduler; to set the affinity the threads to cores mask should be used:

Part of the source code of Fit.cxx

```
179 #ifdef OMP
180     int threadNumberToCpuMap[80];
181 /*    for (int i=0; i<8; i++){
182       threadNumberToCpuMap[2*i+0] = i;
183       threadNumberToCpuMap[2*i+1] = i+8;
184     }*/
185     for ( int iProc = 0; iProc < 4; iProc++ ) {
186         for ( int i = 0; i < 8; i++ ) {
187             threadNumberToCpuMap[2 * i + 0 + iProc * 20] = 4 * i + iProc;
188             threadNumberToCpuMap[2 * i + 1 + iProc * 20] = 4 * i + 32 + iProc;
189         }
190         for ( int i = 0; i < 2; i++ ) {
191             threadNumberToCpuMap[2 * i + 0 + 16 + iProc * 20] = 4 * i + iProc +
    64;
192             threadNumberToCpuMap[2 * i + 1 + 16 + iProc * 20] = 4 * i + 8 +
    iProc + 64;
193         }
194     }
```

3. go to the folder hltsse/TimeHisto and run the bash script, which measures the scalability of the program: “. make_data_omp.sh”; check the scalability:

"root –l make_timehisto_stat_complex.C";

4. set static scheduler without specifying the chunk size, with chunk size 10, 100 and 1000, check the scalability;
5. set dynamic scheduler without specifying the chunk size (and with chunk of size 10 and 100);
6. set guided scheduler without specifying the chunk size (and with chunk of size 10);
7. change the mode of speed measurement: each thread will measure its own speed, then the speed will be added: for this uncomment the line 2 in Fit.cxx; remeasure the scalability with the static scheduler (without chunk size and with chunk of size 10);
8. remeasure the scalability without CPU affinity and without specifying a scheduler; change the mode of the speed measurement (comment line 2 in Fit.cxx) and remeasure the scalability.

5_OMP/4_CBM_KF: solution

1. To parallelise the program we should parallelise the data copying:

Part of the source code of hltsse_solution/Fit.cxx

```
208     #pragma omp parallel reduction(+:NTracks) num_threads(tasks)
209     {
210 //#ifdef OMP
211 //        int s;
212 //        cpu_set_t cpuset;
213 //        int cpuId = threadNumberToCpuMap[omp_get_thread_num()];
214 //        pthread_t thread  = pthread_self();
215 //        CPU_ZERO( &cpuset );
216 //        CPU_SET( cpuId, &cpuset );
217 //        s = pthread_setaffinity_np( thread, sizeof( cpu_set_t ), &cpuset );
218 //        if ( s != 0 ) {
219 //            cout << " pthread_setaffinity_np " << endl;
220 //            handle_error_en( s, "pthread_setaffinity_np" );
221 //        }
222 //#endif
223 
224         #pragma omp for
225         for ( int j = 0; j < tasks; ++j ) {
226             for ( int i = 0; i < NCopy; ++i ) {
227                 if ( j * NCopy + i > MaxNTracks ) {
228                     continue;
229                 }
230                 vTracks[j * NCopy + i] = vTracks[i];
231                 vMCTracks[j * NCopy + i] = vMCTracks[i];
232                 #pragma omp atomic
233                 NTracks++;
234             }
235         }
236     }
```

and fitting itself:

Part of the source code of hltsse_solution/Fit.cxx

```
461   #pragma omp parallel num_threads(tasks) private(iV,ifit)
      firstprivate(NTracksV,NStations, nFittedTracks,timerLocal)
462   {
463     timerLocal.Start();
464     #pragma omp for nowait
465     for( iV=0; iV<NTracksV; iV++ ){ // loop on set of 4 tracks
466       for( ifit=0; ifit<NFits; ifit++){
```

Part of the source code of hltsse_solution/Fit.cxx

```
467           Fit( TracksV[iV], vStations, NStations );
468         }
469         nFittedTracks += vecN;
470       }
471       timerLocal.Stop();
472       double cpuTimeLocal = timerLocal.CpuTime();
473       double realTimeLocal = timerLocal.RealTime();
474
475       if(fabs(cpuTimeLocal)<1.e-8) cpuTimeLocal = realTimeLocal;
476
477   #ifdef OMP
478       int curThredNum = omp_get_thread_num();
479   #else
480       int curThredNum = 0;
481   #endif
482       if(cpuTimeLocal > 0)
483         fitSpeedCPU[curThredNum] = nFittedTracks/cpuTimeLocal;
484       else
485         fitSpeedCPU[curThredNum] = 0;
486
487       if(realTimeLocal > 0)
488         fitSpeedReal[curThredNum] = nFittedTracks/realTimeLocal;
489       else
490         fitSpeedReal[curThredNum] = 0;
491     }
```

variables iV and ifit should be declared private, NTracksV, NStations, nFittedTracks and timerLocal should be initialised - therefore they are declared firstprivate.

2. Since threads are created in the first parallel region, the affinity can be set in the data copying region:

Part of the source code of hltsse_solution/Fit.cxx

```
208       #pragma omp parallel reduction(+:NTracks) num_threads(tasks)
209       {
210   #ifdef OMP
211           int s;
212           cpu_set_t cpuset;
213           int cpuId = threadNumberToCpuMap[omp_get_thread_num()];
214           pthread_t thread  = pthread_self();
215           CPU_ZERO( &cpuset );
216           CPU_SET( cpuId, &cpuset );
217           s = pthread_setaffinity_np( thread, sizeof( cpu_set_t ), &cpuset );
218           if ( s != 0 ) {
219               cout << " pthread_setaffinity_np " << endl;
220               handle_error_en( s, "pthread_setaffinity_np" );
221           }
222   #endif
```

3. As a result after running bash script and root macro we get a picture with a scalability of the program (Fig. 1). The scalability is a straight line, but with hyper threading it should be a stair-like in the ideal case. The reason why it is exactly straight is in the speed measurement mode1 and a static scheduler. The static scheduler distributes tasks equally between threads, so all threads have the same amount of the input data. When running even number of threads the very last thread will be running alone on its physical core. In this case it will finish earlier (on different CPUs from 30% to 60%). But since we measure the execution time of the whole program, it will be determined by the slowest thread. And in such configuration adding one even thread will not give the maximum speedup to the whole program.

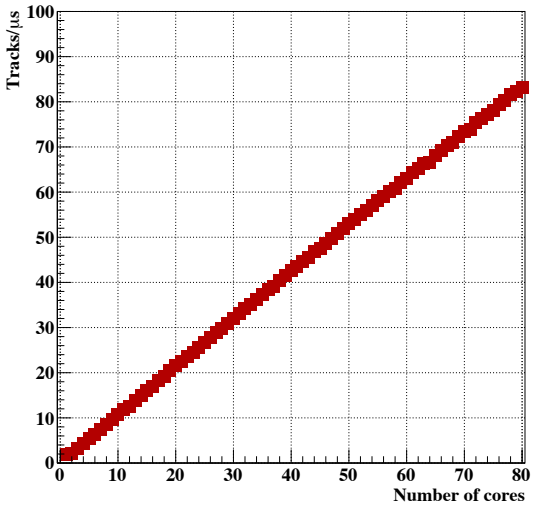


Fig. 1. With CPU affinity, no scheduler.

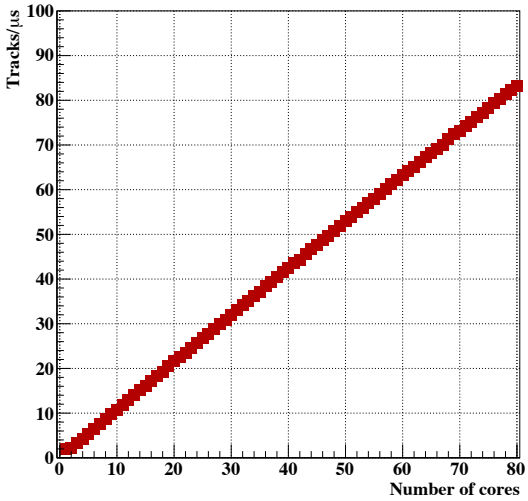


Fig. 2. With CPU affinity, static scheduler.

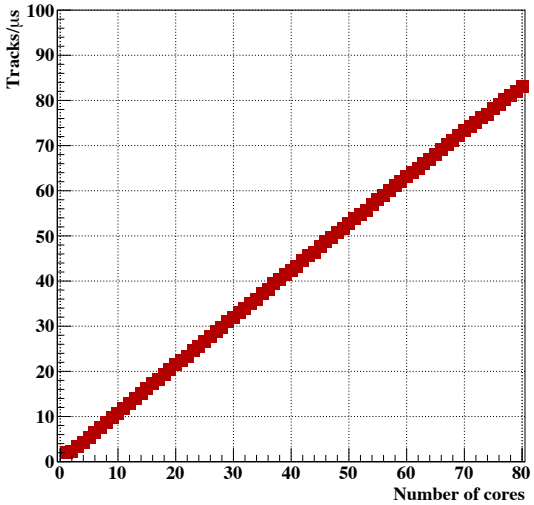


Fig. 3. With CPU affinity, static scheduler, chunk 10.

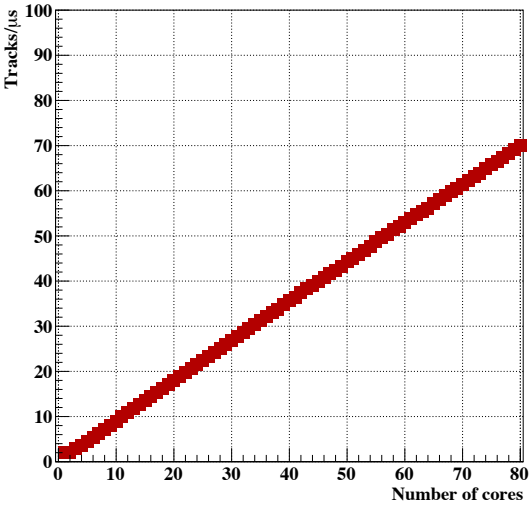


Fig. 4. With CPU affinity, static scheduler, chunk 100.

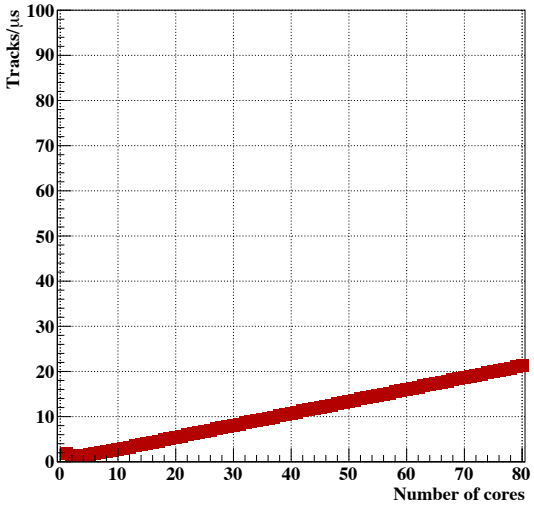


Fig. 5. With CPU affinity, static scheduler, chunk 1000.

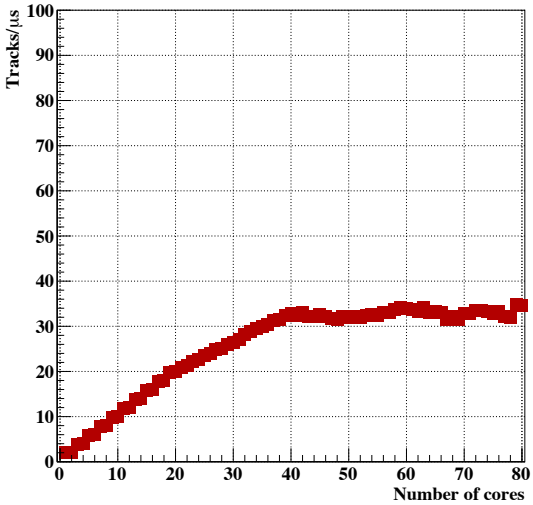


Fig. 6. With CPU affinity, dynamic scheduler.

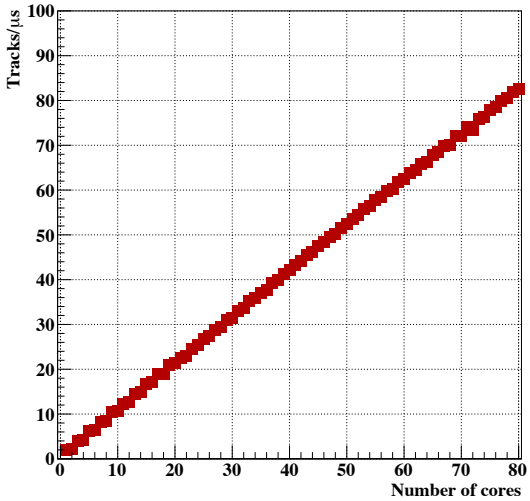


Fig. 7. With CPU affinity, dynamic scheduler, chunk 10.

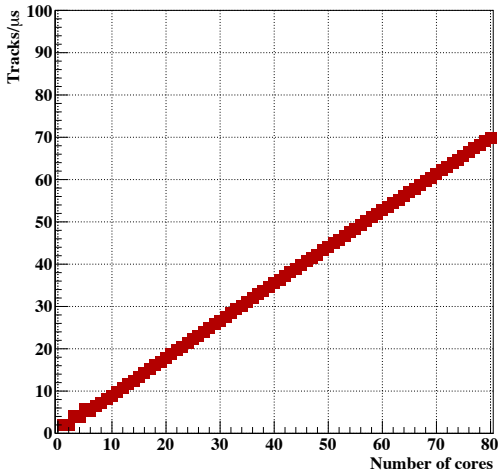


Fig. 8. With CPU affinity, dynamic scheduler, chunk 100.

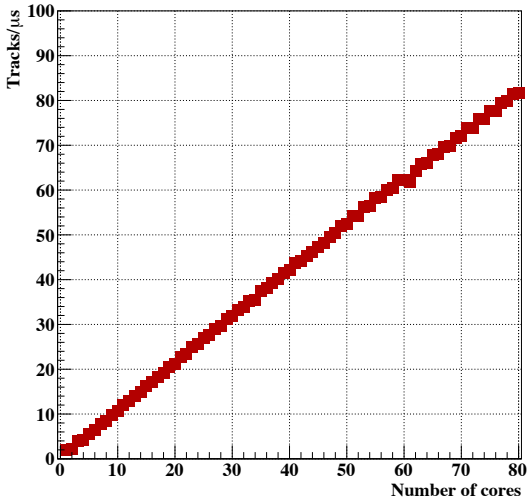


Fig. 9. With CPU affinity, guided scheduler.

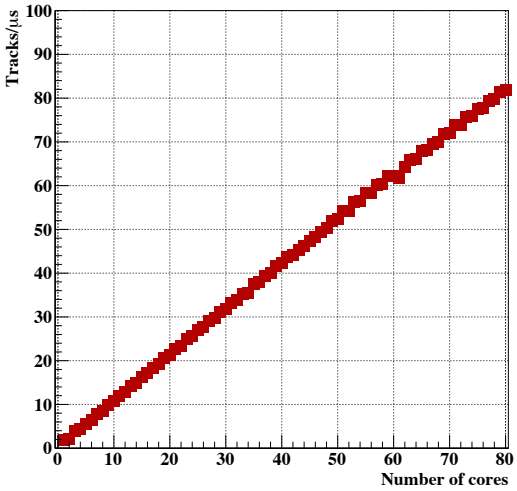


Fig. 10. With CPU affinity, guided scheduler, chunk 10.

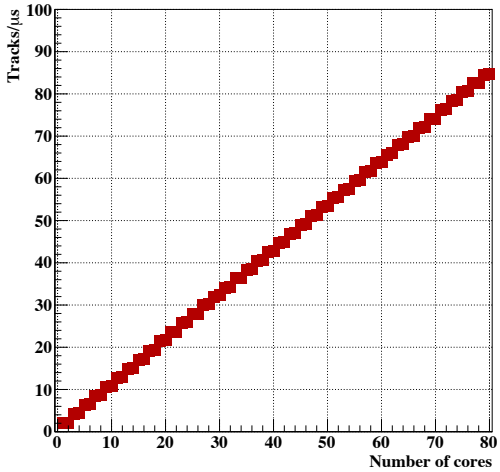


Fig. 11. With CPU affinity, static scheduler, speed measurement mode2.

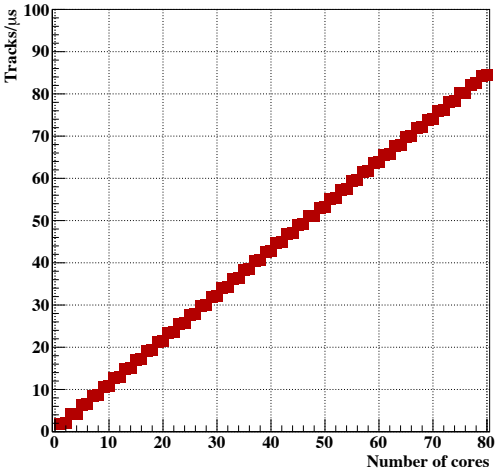


Fig. 12. With CPU affinity, static scheduler, chunk 10, speed measurement mode2.

4. Specifying the static scheduler will not change the picture (Fig. 2), since the default scheduler is static and the chunk size is 1. Increasing the chunk size to 10 (Fig. 3) also will not change the picture, since the size of the input array is divisible by 10. Increasing the chunk size to 100 (Fig. 4) will lead to the drop of the overall speed, since we have 1000 tracks/thread in the input array, which means 250 vector-track per thread. This means, that every second thread will be idle when the last portions will be distributed. Increasing chunk size to 1000 (Fig. 5) will increase this effect even more, since now only every 4th thread is working and the overall speed will drop be a factor of 4.
5. Changing the scheduler to dynamic will change the picture completely (Fig. 6): the linear scalability is destroyed because the dynamic scheduler with a default chunk size of 1 introduces an overhead waiting for a request from threads for a new portion of data. However, on a small amount of cores we observe a stair-like scalability as it should be on a CPU with a hyper threading technology. Increasing the chink size to 10 (Fig. 7) will decrease an overhead and the stair-like structure will be saved. Increasing the chunk size even more (to 100, see Fig. 8) decrease the performance, exactly like it was for a static scheduler, and the stair-like structure disappears.
6. Since guided scheduler is something in between static and dynamic, the pictures for the default chunk of 1 (Fig. 9) and of 10 (Fig. 10) will be also in between: the distance between odd and even points is more smooth, than for dynamic scheduler.
7. When changing the speed measurement mode to the sec on one, the line become stair-like even for a static scheduler with chunk size 1 (Fig. 11). This means, that the speed of the threads is absolutely equal and really, the last even thread is waiting for all others in the end of the program. Changing the chunk size to 10 (Fig. 12) will not change the pictures for the static scheduler.
8. Without CPU affinity the picture is changed (Fig. 13): it is divided into two sections. The first one is sharper, since in the system numeration first logical cores correspond to different physical cores. When two threads are running on the same physical cores, the slope of the line is smaller. When changing the speed measurement mode to the first again (Fig. 14), the drop in the overall speed appears after the thread 36: it is caused by hyper threading, since the speed is determined by the slowest thread and appears when two threads are running on the same physical core the first time.

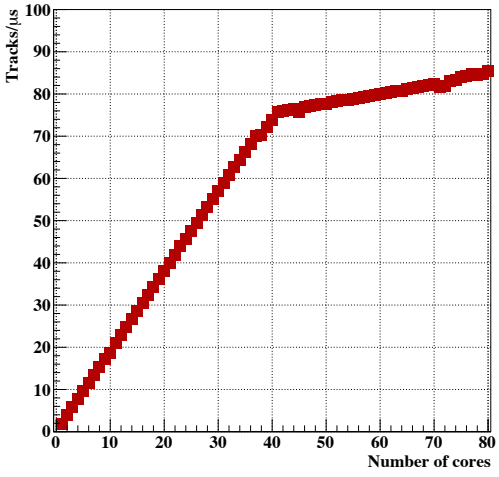


Fig. 13. Without CPU affinity, static scheduler, speed measurement mode2.

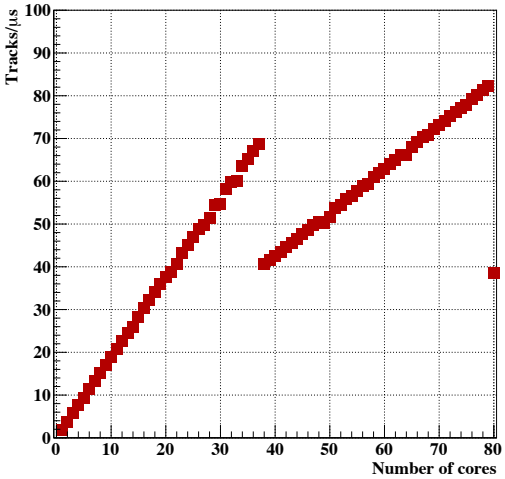


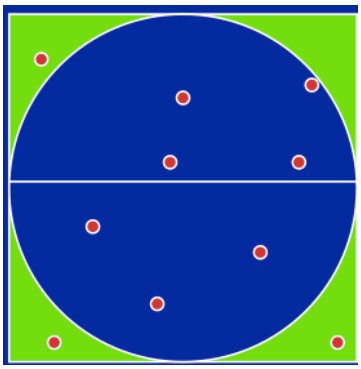
Fig. 14. Without CPU affinity, static scheduler, speed measurement mode1.

5_OMP/5_MonteCarlo: description

The generally accepted key performance index for parallel programs is speed-up factor, which is defined as serial execution time on one core, divided by parallel execution time on N cores. Naively one would expect program running on N cores to be N times faster, which is called linear scalability. However due to

the normal presence of some small scalar parts in the program usually its not the case and one will see some slowing down with the increase of cores number. The aim of this exercise is to parallelise the Monte Carlo π calculation and measure and explain its scalability.

Monte Carlo method is a general approach to solve tough problems with the use of random numbers. In our exercise we will compute pi value with a digital dart board. It's not the most precise way to get pi value, although its a nice example for scalability measurement with a simple calculations.



In a scalar version we throw a random darts with coordinates inside the area of a square of a size 1x1. If the coordinates are truly random the probability for dart to fall inside the circle is proportional to its area: $P = \pi R^2$; The probability of falling inside the square is one by default. Thus, one can compute π value with an expression:

$\pi = 4\ Ncircle / Nsquare$,
here Ncircle is the number of points inside the circle, Nsquare is a total number of points.

One is supposed to rewrite the program in parallel which randomly chooses points inside the square, calculates the fraction of points inside the circle and calculates π.
In order to run program type :

g++ MonteCarlo.cpp -O3 -fopenmp -o MonteCarlo.out; ./MonteCarlo.out n 1,
where n is number of cores you want to use.
In order to calculate scalability you can use the script:
. run_scalability.sh
And draw the speed-up factor dependence:
root -l make_timehisto_stat_complex.C

5_OMP/5_MonteCarlo: solution

Setting thread to core affinity for scalability measurements

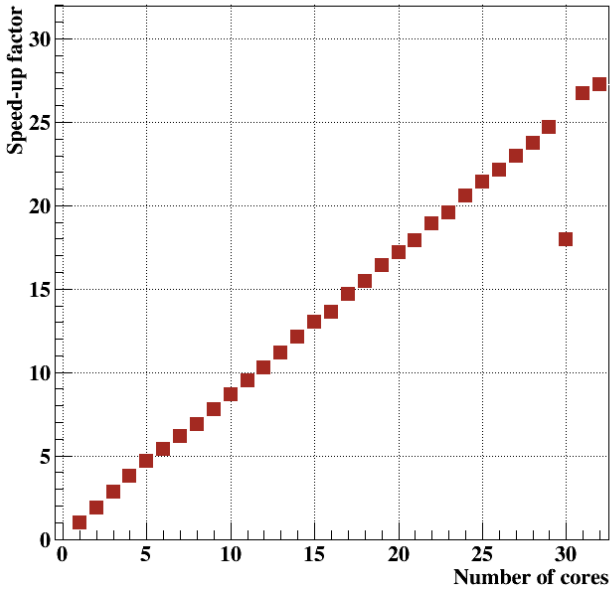


Fig. 15. Scalability of the pi estimation.

In general case exact thread to core correspondence is not defined by default. It may happen that in order to optimise resource usage processor sends thread from one to core to the other. Such a thread jump takes time and may result in overhead and spoil scalability of an application. In order to avoid it and improve performance one needs to set thread to core affinity. For this purpose one can use Pthread API. The **pthread_setaffinity_np()** function sets the CPU affinity mask of the thread **thread** to the CPU set pointed to by **cpuset**. As a result of this function the thread will be running on one of the CPUs in **cpuset**. The argument **cpusetsize** is the length (in bytes) of the buffer pointed to by **cpuset**. Typically, this argument would be specified as **sizeof(cpu_set_t)**.

This kind of problem is intrinsically parallel, since the loop iterations are practically independent.

One can convert scalar version to parallel just by adding few lines of code, paying attention to keeping some variables private instead of shared and using reduction clause to calculate the increment in a manner:

#pragma omp parallel for private(x,y,test, seed) reduction(+:Ncirc)

Now each thread will operate with its own copy of **x, y, test** and **seed** variables making code thread-safe, as well as increment of **Ncirc** will be sumed up to one global variable in the end.

The final scalability after setting affinity and making loop thread-safe one can get output like in the Fig. 1.

HPC Practical Course
Part 3.2

Intel Threading Building Blocks (ITBB)

V. Akishina, I.Kisel,
I. Kulakov, M. Zyzak

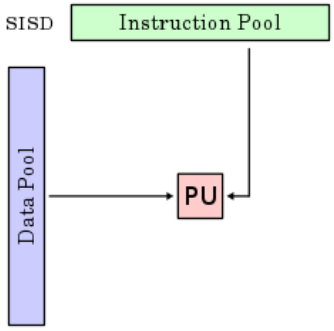
Goethe University of Frankfurt am Main

09 Jul 2014

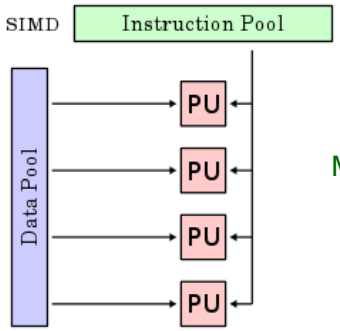
Based on the official ITBB tutorial

Computer Architectures

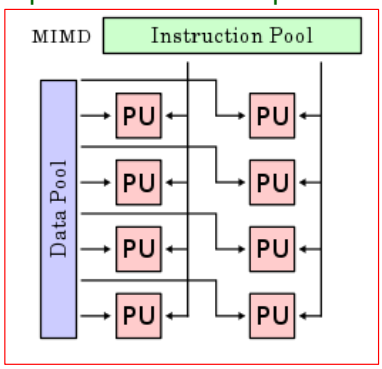
Single Instruction Single Data



Single Instruction Multiple Data



Multiple Instruction Multiple Data



Taken from: http://en.wikipedia.org/wiki/Flynn's_taxonomy

Content

- Parallel_for
- Parallel_reduce
- Dependent threads: mutual exclusion, atomic operations
- Pipeline
- Exercises

Parallel_for

Scalar

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

independent iterations

ITBB

```
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
    my_a(a)
    {}
};
```

all needed data are kept here

actual work is done here

```
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

create tasks and distribute them between cores

Parallel_reduce

Scalar

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

ITBB

```
float my_sum;
void operator()( const blocked_range<size_t>& r ) {
    float *a = my_a;
    float sum = my_sum;
    size_t end = r.end();
    for( size_t i=r.begin(); i!=end; ++i )
        sum += Foo(a[i]);
    my_sum = sum;
}

SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0) {}

void join( const SumFoo& y ) {my_sum+=y.my_sum;}
```

result of each task

actual work is done here

split constructor

accumulate result

```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce( blocked_range<size_t>(0,n), sf );
    return sf.my_sum;
}
```

create tasks and distribute them between cores

Mutual Exclusion: Scoped_lock

shared variable

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```

it is important to keep FreeList unchanged by other threads here

block this the scope for all the threads, except the current one

Atomic Operations

Problem:

```
int x;
…

x--;
if ( x == 0 ) Finilize();
```

temporary copy

Tread A	Tread B
t_a	
	t_b
x = t	
	x = t
if (x == 0)	
	if (x == 0)

x -> x – 1 (not x – 2 !)

unnecessary 2nd call

Solution:

```
atomic<int> x;
…

if ( x-- == 0 ) Finilize();
```

this line can be called only by one tread at one moment

Tread A	Tread B
if (x-- == 0) Finilize();	
	if (x-- == 0) Finilize();

Pipeline

Read chunk from file → Square numerals in chunk → Write chunk to output file

```
//! Filter that changes each decimal number to its square.
class MyTransformFilter: public tbb::filter {
public:
    MyTransformFilter();
    void* operator()( void* item );
};

MyTransformFilter::MyTransformFilter() :
    tbb::filter(parallel)
{}

void* MyTransformFilter::operator()( void* item ) {
    TextSlice& input = *static_cast<TextSlice*>(item);
    // Add terminating null so that strtol works right even if number is at end of the input.
    *input.end() = '\0';
    char* p = input.begin();
    TextSlice& out = *TextSlice::allocate( 2*MAX_CHAR_PER_INPUT_SLICE );
    char* q = out.begin();
    for(;;) {
        while( p<input.end() && !isdigit(*p) )
            *q++ = *p++;
        if( p==input.end() )
            break;
        long x = strtol( p, &p, 10 );

        long y = x*x;
        sprintf(q,"%ld",y);
        q = strchr(q,0);
    }
    out.set_end(q);
    input.free();
    return &out;
}
```

the work is done here

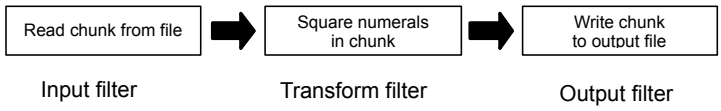
data from the previous filter

allocate output data

deallocate output data

give data to the next filter

Pipeline: Continue




```
// Create the pipeline
tbb::pipeline pipeline;

// Create file-reading writing stage and add it to the pipeline
MyInputFilter input_filter( input_file );
pipeline.add_filter( input_filter );

// Create squaring stage and add it to the pipeline
MyTransformFilter transform_filter;
pipeline.add_filter( transform_filter );

// Create file-writing stage and add it to the pipeline
MyOutputFilter output_filter( output_file );
pipeline.add_filter( output_filter );

// Run the pipeline
tbb::tick_count t0 = tbb::tick_count::now();
// Need more than one token in flight per thread to keep all threads
// busy; 2-4 works
pipeline.run( nthreads*4 );
tbb::tick_count t1 = tbb::tick_count::now();
```

create the pipeline

add all filters consecutively

run the pipeline on certain number of threads

Exercises

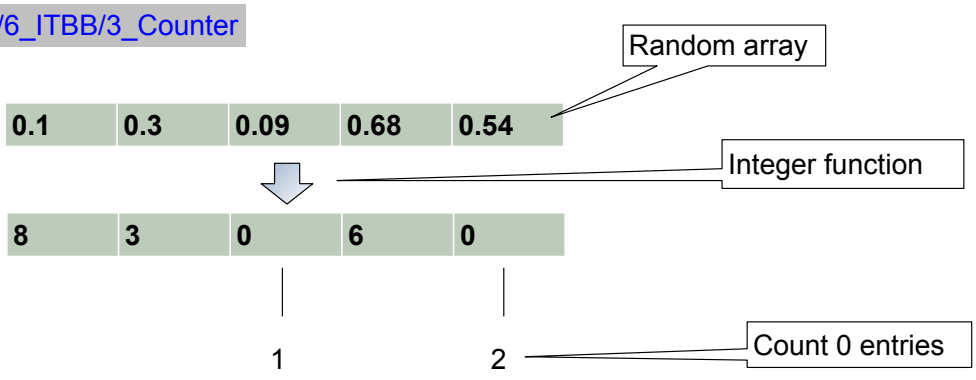
Run AddPath.sh in order to make compiler know about ITBB and Vc

Exercises/6_ITBB/1_Matrix

Exercises/6_ITBB/2_Pi

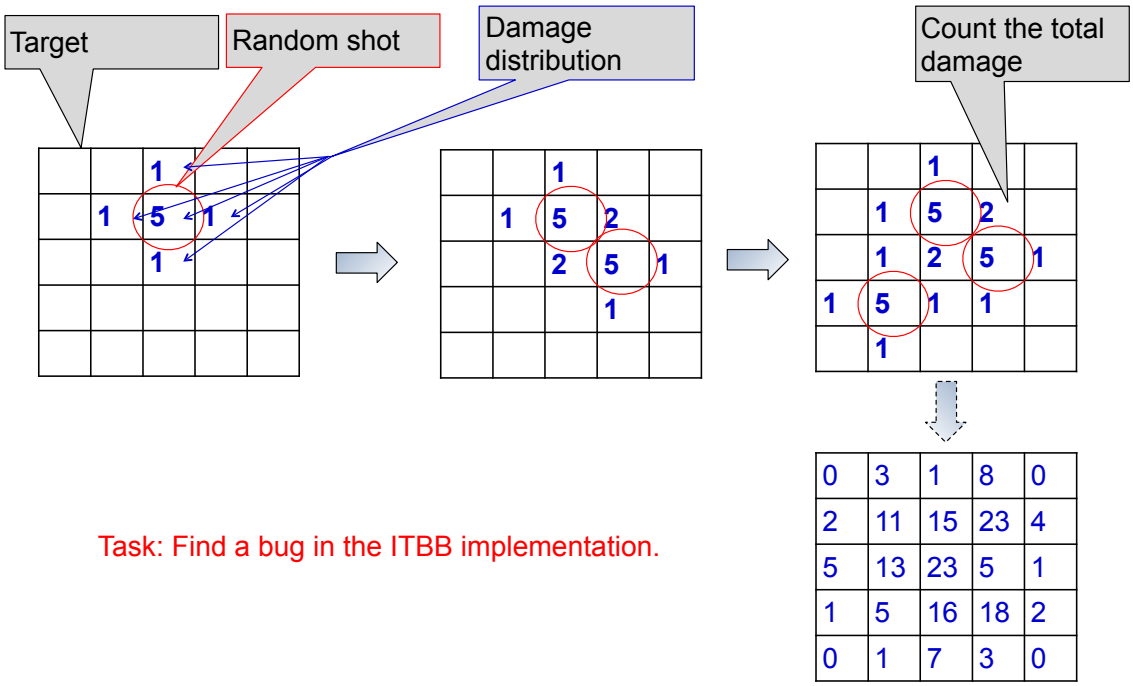
See the previous SIMD and OpenMP exercises.
Task: Implement with ITBB

Exercises/6_ITBB/3_Counter



Exercises

Exercises/6_ITBB/4_Target

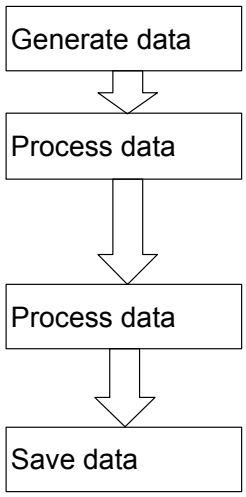


Task: Find a bug in the ITBB implementation.

Exercises

Exercises/6_ITBB/5_Pipeline

```
TStopwatch timer;
for( int j = 0; j < NChunks; ++j ) {
  DataChunk data;

    // generate data
  for( int i = 0; i < ChunkSize; ++i )
    data.a[i] = float(rand())/float(RAND_MAX); // from 0 to 1

    // make some processing
  for( int i = 0; i < ChunkSize; ++i ) {
    float &x = data.a[i];
    for( int i = 0; i < 100; ++i )
      x = sin(x);
  }

    // another processing
  for( int i = 0; i < ChunkSize; ++i ) {
    float &x = data.a[i];
    for( int i = 0; i < 100; ++i )
      x = tan(x);
  }

    // write the result in the file
  for( int i = 0; i < ChunkSize; ++i )
    file << data.a[i] << endl;
}

timer.Stop();
```



Task: Parallelize with ITBB.

Kalman Filter Track Fit for CBM Experiment

Exersices/6_ITBB/6_CBM_KF

Compile

make single (or “make tbb”)

Run

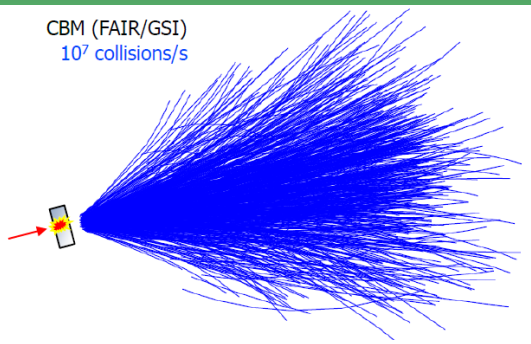
./single (or “./tbb [nThreads to run]”)

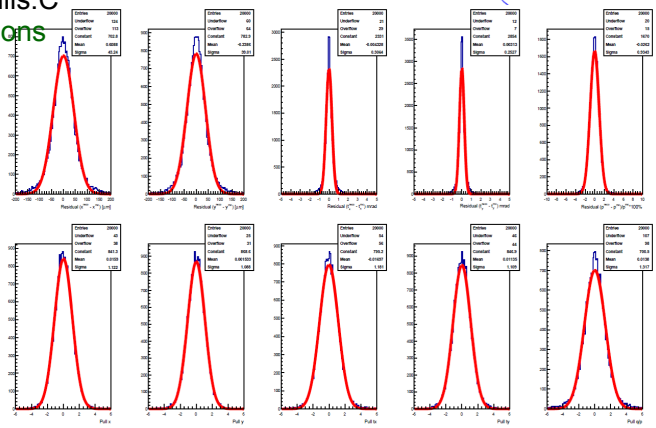
Check results

cd QualityHisto

root –l –q histo_particle.C; root –l Pulls.C

(keep attention to sigma of distributions and number of entries)





Task:

Parallelize using ITBB

3.2. Intel Threading Building Blocks

Exercises are located at Exercises/6_ITBB/
Solutions are located at Exercises/6_ITBB/***/***_solution.cpp
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

6_ITBB/1_Matrix: description

The Matrix exercise requires to parallelize the square root extraction over a set of float variables arranged in a square matrix using both the ITBB and the Vc library. The initial code implements scalar and vector parts using the Vc library, the space for the ITBB+Vc implementation is blank. Therefore the initial output shows 0 time for vector calculations and an infinite speed up factor, that should be currently ignored.

Part of the source code of Matrix.cpp

```
34 template<typename T> // required calculations
35 T f(T x) {
36   return sqrt(x);
37 }
…
52 class ApplyTBB{ // TODO: finish
53 public:
54
55   void operator()(const blocked_range<long unsigned int> &range, int cpuId =
-1) const {
56   }
57
58 //   ApplyTBB( float a_[N][N],  float c_[N][N]): a(a_), c(c_){}
59   ~ApplyTBB(){}
60 };
…
82    /// Vc
83   TStopwatch timerVc;
84   for( int ii = 0; ii < NIter; ii++ )
85     for( int i = 0; i < N; i++ ) {
86       for( int j = 0; j < N; j+=float_v::Size ) {
87           float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
88           float_v &cVec = (reinterpret_cast<float_v&>(c_simd[i][j]));
89           cVec = f(aVec);
90       }
91     }
92   timerVc.Stop();
93
94    /// ITBB and Vc
95   TStopwatch timerITBB;
96   for( int ii = 0; ii < NIter; ii++ )
97   {
98      // TODO
99   }
```

Part of the source code of Matrix.cpp

```
100   timerITBB.Stop();
```

Typical output

```
Time scalar:  798.12 ms
Time Vc:      200.525 ms, speed up 3.98015
Time timerITBB:  0 ms, speed up inf
Parallel and scalar results are the same.
ERROR! Parallel and scalar results are not the same.
```

6_ITBB/1_Matrix: solution

In order to parallelize calculations we need to provide the **ApplyTBB** class input and output arrays, therefore a constructor, which includes pointers to these arrays, should be created. The **operator()** is filled with the required calculations. The most natural way to distribute calculations between cores is parallelization of the outer loop, which must be changed to a loop over **blocked_range**.

Simple parallel_for can be used to create and run the tasks on different cores.

Part of the source code of Matrix_solution.cpp

```
54  class ApplyTBB{
55
56  public:
57
58    float (*a)[N];
59    float (*c)[N];
60
61    void operator()(const blocked_range<long unsigned int> &range, int cpuId =
    -1) const {
62      for(int i = range.begin(); i != range.end(); ++i){
63        for( int j = 0; j < N; j+=float_v::Size ) {
64            float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
65            float_v &cVec = (reinterpret_cast<float_v&>(c[i][j]));
66            cVec = f(aVec);
67        }
68      }
69    }
70
71    ApplyTBB( float a_[N][N],  float c_[N][N]): a(a_), c(c_){}
72
73    ~ApplyTBB(){
74    }
75  };
  …
109    /// ITBB
110   TStopwatch timerITBB;
111   for( int ii = 0; ii < NIter; ii++ )
112   {
113     parallel_for(blocked_range<size_t>(0,N), ApplyTBB(a,c_tbb));
114   }
115   timerITBB.Stop();
```

Typical output after solution

```
Time scalar:  798.184 ms
Time Vc:      200.676 ms, speed up 3.97748
Time timerITBB:  65.238 ms, speed up 12.235
SIMD and scalar results are the same.
SIMD and scalar results are the same.
```

An expected speed up from the parallelization on a computer with 40 hyper-threaded cores is about 50. The obtained speed up is 3, that is much less. A possible reason is that calculations are extremely simple and fast.

To check it, one can increase the amount of calculations in **f()** function, this leads to increase of parallelisation speed up:

Part of the source code of Matrix.cpp

```
54 template<typename T> // required calculations
55 T f(T x) {
56   T r = x;
57   for( int i = 0; i < 20; i++ )
58     r = sqrt(r);
59   return r;
60 }
```

Typical output after solution

```
Time scalar:  16779.2 ms
Time Vc:      3987.25 ms, speed up 4.20821
Time timerITBB:  164.553 ms, speed up 101.968
SIMD and scalar results are the same.
SIMD and scalar results are the same.
```

6_ITBB/2_Pi: description

The Pi program provides a scalar code to calculate the value of π by numerical integration. It must be run on different cores using ITBB.

Part of the source code of pi.cpp

```
26   int i;
27   double x, pi, sum = 0.0;
28   double start_time, run_time;
29
30   step = 1.0/(double) num_steps;
31
32 //   task_scheduler_init init(1); // run 1 thread only
33
34   TStopwatch timer;
35
36   // TODO parallelize the loop
37   for(int i = 1; i != num_steps; ++i){
38     const double x = (i-0.5)*step;
39     sum += 4.0/(1.0+x*x);
40   }
41
42   pi = step * sum;
```

Typical output

```
pi with 2000000000 steps is 3.141593 in 19.477509 seconds
```

6_ITBB/2_Pi: solution

To parallelize this task one needs to use parallel_reduce to split the sum between different cores and then joining together the obtained results. **ApplyTBB** class needs to include only the output variable, **operator()** body - the integral calculations, and **join()** function - the simple sum.

Part of the source code of pi.cpp

```
26 class ApplyTBB{
27
28   double fStep;
29
30 public:
31   double sum;
32
33   void operator()(const blocked_range<long unsigned int> &range, int cpuId =
   -1) {
34     for(int i = range.begin(); i != range.end(); ++i){
35       const double x = (i-0.5)*step;
36       sum += 4.0/(1.0+x*x);
37     }
38   }
39
40   ApplyTBB( ApplyTBB& x, split ) : fStep(x.fStep), sum(0) {}
41   void join( const ApplyTBB& y ) { sum += y.sum;}
42
43   ApplyTBB(double step_):fStep(step_), sum(0){}
44   ~ApplyTBB(){}
45 };
46
47
48 int main ()
49 {
50   int i;
51   double x, pi, sum = 0.0;
52   double start_time, run_time;
53
54   step = 1.0/(double) num_steps;
55
56 //   task_scheduler_init init(1); // run 1 thread only
57
58   TStopwatch timer;
59
60   ApplyTBB at(step);
61   parallel_reduce( blocked_range<size_t>(1,num_steps), at );
62
63   pi = step * at.sum;
```

Typical output after solution

```
pi with 2000000000 steps is 3.141593 in 0.521898 seconds
```

The obtained speed up on 40 hyper threaded cores is 37 times.

6_ITBB/3_Count: description

The Count exercise implements a counting process. Input floating point data is transformed to integer data, then one needs to count how many 0 entries are in the whole data array.

This task mast be parallelized using ITBB in two different ways:

1. Using mutex to lock a region of the code.
2. Using atomic operations to lock variables.

Part of the source code of count.cpp

```
28 int ComplicatedFunction( float x ){ // just to simulate some time-consuming
   calculations, which can be parallelized
29   return (int)( cos(sin(x*3.14)) * 10 - 5 );
30 }
 …
63     // fill classes by random numbers
64   for( int i = 0; i < N; i++ ) {
65     a[i] = (float(rand())/float(RAND_MAX)); // put a random value, from 0 to 1
66   }
67
68   TStopwatch timer;
69   for(int i = 0; i != N; ++i){
70     if (ComplicatedFunction(a[i]) == 0) counter++;
71   }
72   timer.Stop();
```

Typical output

```
 Scalar counter:     489566 Time: 161.237961 ms.
 TBB atomic counter: 0 Time: 0.000000 ms.
 TBB mutex counter:  0 Time: 0.000954 ms.
```

6_ITBB/3_Count: solution

The parallelization is achieved by distribution of loop iterations over different threads. In this case **counter++** operation makes iterations dependent on each other and when it is performed on one thread all other threads must wait.

1. To implement it using atomic feature one needs to declare **counter** variable simple as **atomic<int>**.
2. To implement it with mutex feature one needs to declare spin_mutex, lock it before increment and unlock it after decrement. Unlocking can be done automatically when lock is destructed at the end of scope.

Part of the source code of count.cpp

```
21 int counterParM = 0; // parallelization using mutex
22 atomic<int> counterParA; // parallelization using atomic
 …
29 class ApplyTBBA{
30   const float  * const a;
31
32 public:
33   void operator()(const blocked_range<long unsigned int> &range, int cpuId =
   -1) const {
34     for(int i = range.begin(); i != range.end(); ++i){
35       if (ComplicatedFunction(a[i]) == 0) counterParA++;
36     }
```

Part of the source code of count.cpp

```
37    }
38
39    ApplyTBBA(const float * const a_):a(a_){}
40    ~ApplyTBBA(){}
41  };
42
43  spin_mutex Mutex;
44
45  class ApplyTBBM{
46    const float * const a;
47
48  public:
49    void operator()(const blocked_range<long unsigned int> &range, int cpuId =
    -1) const {
50      for(int i = range.begin(); i != range.end(); ++i){
51        if (ComplicatedFunction(a[i]) == 0) {
52          spin_mutex::scoped_lock lock(Mutex);
53          counterParM++;
54        }
55      }
56    }
57
58    ApplyTBBM(const float * const a_):a(a_){}
59    ~ApplyTBBM(){}
60  };
 …
81    timer.Start();
82    parallel_for( blocked_range<size_t>(0,N), ApplyTBBA(a) );
83    timer.Stop();
84
85    float timeParA = timer.RealTime()*1000;
86
87    timer.Start();
88    parallel_for( blocked_range<size_t>(0,N), ApplyTBBM(a) );
89    timer.Stop();
```

Typical output after solution

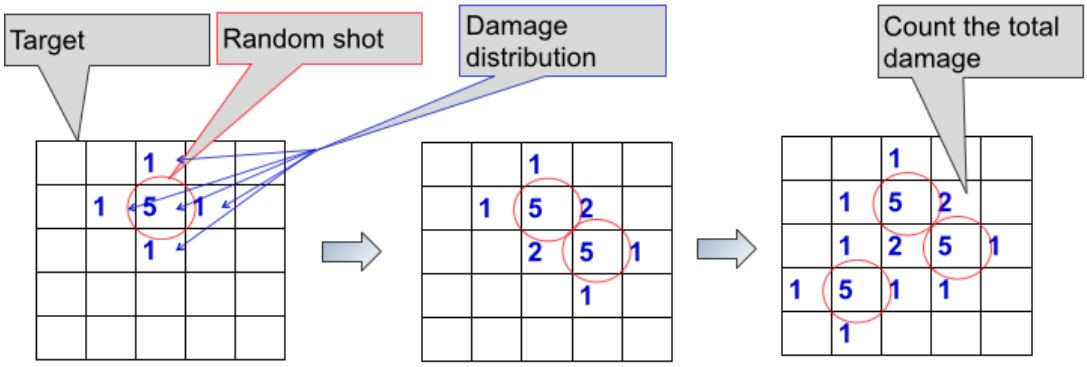
```
Scalar counter:     489566 Time: 159.914978 ms.
TBB atomic counter: 489566 Time: 76.969864 ms.
TBB mutex counter:  489566 Time: 298.513885 ms.
```

The resulting speed up with atomic is much less than the number of cores, but this one can expect, since the threads must wait for each other. The resulting speed up with locks is even smaller, since creation and destruction of lock takes additional time.

6_ITBB/4_Target: description

In the Target exercise a distribution of damages made to a wall is calculated. The wall plane is divided into cells and the damage done in each cell must be calculated. Damaging is done in shots, each shot produces 5 damage of the cell, 1 in each of the four neighboring cells, and 0 to all others.

A version of the code, which implements the procedure is given. It is already parallelized using ITBB, but works either slow or can not finish the calculations at all.
One needs to find a reason to such behavior and fix it with as small changes in the code as possible.



Part of the source code of target.cpp

```
22 const int Size = 5;
23 const int N = 1000;
24
25 struct TShot {
26   int x,y;
27   float spread;
28 };
29
30 const float PI = 3.1415926f;
31
32 float Response( float x, float spread ) {
33   int s = spread;
34   for( int i = 0; i < 100000; ++i ) s = sin(s); // simulate complicated
   computations, which would be in a real life response function
35   return 1/spread/sqrt(2*PI)*exp(-0.5*x*x/spread/spread)*s;
36 }
37
38 void Update( float& xmm, float& xlm, float& xml, float& xrm, float& xmr, float
   spread ) {
39   xmm += Response( 0, spread );
40   float res1 = Response( 1, spread );
41   xlm += res1;
42   xml += res1;
43   xrm += res1;
44   xmr += res1;
45 }
46
47 spin_mutex mutexes[Size][Size];
48
49 class ApplyTBB{
50   TShot *shots;
51   float (*target)[Size];
52 public:
53
54   void operator()(const blocked_range<long unsigned int> &range, int cpuId =
   -1) const {
55     for(int i = range.begin(); i != range.end(); ++i) {
56       const int x = shots[i].x;
57       const int y = shots[i].y;
58       const float spread = shots[i].spread;
59       {
60         spin_mutex::scoped_lock lock0(mutexes[x][y]);
61         spin_mutex::scoped_lock lock1(mutexes[x-1][y]);
```

Part of the source code of target.cpp

```
62          spin_mutex::scoped_lock lock2(mutexes[x][y-1]);
63          spin_mutex::scoped_lock lock3(mutexes[x+1][y]);
64          spin_mutex::scoped_lock lock4(mutexes[x][y+1]);
65          Update( target[x][y], target[x-1][y], target[x][y-1], target[x+1][y],
    target[x][y+1], spread );
66        }
67      }
68    }
69
70    ApplyTBB(float target_[Size][Size], TShot
    *shots_):target(target_),shots(shots_){}
71    ~ApplyTBB(){}
72  };
73
74  bool CompareResults( float a1[Size][Size], float a2[Size][Size] ) {
75    bool ok = 1;
76    for( int i = 0; i < Size; i++ )
77      for( int i2 = 0; i2 < Size; i2++ ) {
78        ok &= (a1[i][i2] == a2[i][i2]);
79        //printf( " %d %d : %f - %f \n", i, i2, a1[i][i2], a2[i][i2] );
80      }
81    return ok;
82  }
83
84  int main ()
85  {
86    float target[Size][Size];
87    TShot shots[N];
88
89      // prepare data
90    for( int i = 0; i < Size; i++ )
91      for( int i2 = 0; i2 < Size; i2++ )
92        target[i][i2] = 0;
93
94    for( int i = 0; i < N; i++ ) {
95      shots[i].x = float(rand())/float(RAND_MAX)*(Size-2) + 1; // put a random
    value, from 1 to Size-1
96      shots[i].y = float(rand())/float(RAND_MAX)*(Size-2) + 1;
97      shots[i].spread = float(rand())/float(RAND_MAX)*8 + 1; // from 1 to 10
98    }
99
100   TStopwatch timer;
101   for( int i = 0; i != N; ++i ) {
102     const int x = shots[i].x;
103     const int y = shots[i].y;
104     const float spread = shots[i].spread;
105     Update( target[x][y], target[x-1][y], target[x][y-1], target[x+1][y],
    target[x][y+1], spread );
106   }
107   timer.Stop();
108
109   float targetScalar[Size][Size];
```

Part of the source code of target.cpp

```
110       // save result
111   for( int i = 0; i < Size; i++ )
112     for( int i2 = 0; i2 < Size; i2++ )
113       targetScalar[i][i2] = target[i][i2];
114
115   float timeScalar = timer.RealTime()*1000;
116
117   printf( " Scalar Time: %f ms. \n", timeScalar );
118
119     // prepare data
120   for( int i = 0; i < Size; i++ )
121     for( int i2 = 0; i2 < Size; i2++ )
122       target[i][i2] = 0;
123
124 //   task_scheduler_init init(1); // run 1 thread only
125
126   timer.Start();
127   parallel_for( blocked_range<size_t>(0,N), ApplyTBB(target,shots) );
128   timer.Stop();
```

Typical output

```
 Scalar Time: 1062.815186 ms.
[program hangs]
```

6_ITBB/4_Target: solution

The problem with the initial code is a wrong locks order. When both lock1 (for cell x-1) and lock3 (for cell x+1) are locked, after lock0 (for cell x) a dead lock situation is possible. Consider one thread is working with x=5, the other one with x=6. The first thread locks cell 5, the second one - cell 6. Then the first thread tries to lock cell 6, but it is locked by the second one, so the first one must wait until the second thread is finished and unlocks cell 6. Similarly, the second thread needs cell 5 and must wait for it to be unlocked. Both threads will wait for an infinite time.

To avoid such possibility one needs to call locks in the same order by all threads. To achieve this one can sort locks by coordinates:

Part of the source code of target.cpp

```
65         spin_mutex::scoped_lock lock1(mutexes[x-1][y]);
66         spin_mutex::scoped_lock lock2(mutexes[x][y-1]);
67         spin_mutex::scoped_lock lock0(mutexes[x][y]);
68         spin_mutex::scoped_lock lock4(mutexes[x][y+1]);
69         spin_mutex::scoped_lock lock3(mutexes[x+1][y]);
```

Typical output after solution

```
 Scalar Time: 1062.598999 ms.
 TBB Time:    966.996887 ms.
 Results are the same
```

A minor speed up on the 40-core system is achieved now. The reason for that is that the grid size is 5 by 5 (there are only 9 possible positions for shots, most of them mutually exclude each other). When the grid size is increased to 82x82, the speed up factor of 18 is achieved, which is expected taking into account the dependence between the threads and an overhead from locks.

Typical output after solution

```
Scalar Time: 1036.182861 ms.
TBB Time:    57.986977 ms.
Results are the same
```

6_ITBB/5_Pipeline: description

The Pipeline exercise proposes to parallelize a short pipeline, which consists of 4 stages: 1) generating the data, 2) first processing of the data, 3) second processing stage, 4) saving the data. The task is to finish parallelisation of the process by running these 4 stages on different cores using **tbb::pipeline**. Calls of the filters are implemented, one needs to implement properly the filter classes, one per each stage.

Part of the source code of pipeline.cpp

```
45   for( int j = 0; j < NChunks; ++j ) {
46     DataChunk data;
47
48       // generate data
49     for( int i = 0; i < ChunkSize; ++i )
50       data.a[i] = float(rand())/float(RAND_MAX); // from 0 to 1
51
52       // make some processing
53     for( int i = 0; i < ChunkSize; ++i ) {
54       float &x = data.a[i];
55       for( int i = 0; i < 100; ++i )
56         x = sin(x);
57     }
58
59       // another processing
60     for( int i = 0; i < ChunkSize; ++i ) {
61       float &x = data.a[i];
62       for( int i = 0; i < 100; ++i )
63         x = tan(x);
64     }
65
66       // write the result in the file
67       file << data.a[i] << endl;
68       file << data.a[i] << endl;
69   }
```

Typical output

```
Time:    776.716003 ms.
```

6_ITBB/5_Pipeline: solution

The data is already divided into chunks, therefore we can create chunks one by one, that makes parallelization between chunks possible. The **InputFilter** class must implement **operator()**, which allocate memory for chunk of data, fills it and returns a pointer to the allocated memory. Also one needs to check the number of chunks processed to stop eventually. Since this is the very first filter, the argument is not used.

The **Process1Filter operator()** receives data chunk as an argument, then it needs to be processed and returned to the next filter. The processing stage can be put as a member-function for clearness. The **Process2Filter** is implemented in the same way. The **OutputFilter** writes each data in the chunk into a file and then returns 0, since this is the final stage.

Part of the source code of pipeline.cpp

```
37 class InputFilter: public tbb::filter {
38 public:
39     InputFilter(): filter(serial_in_order){}
40     ~InputFilter(){}
41 private:
42     void* operator()(void*);
43 };
44
45 void* InputFilter::operator()(void*) {
46
47   if (++nChunks > NChunks) return 0;
48
49   DataChunk* data = new DataChunk;
50
51     // generate
52   for( int i = 0; i < ChunkSize; ++i )
53     data->a[i] = float(rand())/float(RAND_MAX); // from 0 to 1
54
55 #ifndef MUTE
56   cout << " ChunkCreated " << endl;
57 #endif
58   return data;
59 }
60
61   /// ---------------------------------
62
63 class Process1Filter: public tbb::filter {
64 public:
65     Process1Filter(): filter(serial_in_order){}
66     ~Process1Filter(){}
67 private:
68     void* operator()(void*);
69
70     void Core(float& x){
71       for( int i = 0; i < 100; ++i )
72         x = sin(x);
73     }
74 };
75
76 void* Process1Filter::operator()(void* d) {
77   DataChunk* data = static_cast<DataChunk*>(d);
78
79   for( int i = 0; i < ChunkSize; ++i )
80     Core(data->a[i]);
81
82   return data;
83 }
84
85   /// ---------------------------------
86
87 class Process2Filter: public tbb::filter {
88 public:
```

Part of the source code of pipeline.cpp

```
89      Process2Filter(): filter(serial_in_order){}
90      ~Process2Filter(){}
91  private:
92      void* operator()(void*);
93
94      void Core(float& x){
95        for( int i = 0; i < 100; ++i )
96          x = tan(x);
97      }
98  };
99
100 void* Process2Filter::operator()(void* d) {
101   DataChunk* data = static_cast<DataChunk*>(d);
102
103   for( int i = 0; i < ChunkSize; ++i )
104     Core(data->a[i]);
105
106   return data;
107 }
108
109   /// ---------------------------------
110
111 class OutputFilter: public tbb::filter {
112 public:
113     OutputFilter(ofstream& file_): filter(serial_in_order), fFile(file_){}
114     ~OutputFilter(){}
115 private:
116   ofstream& fFile;
117   void* operator()(void*);
118 };
119
120 void* OutputFilter::operator()(void* d) {
121   DataChunk* data = static_cast<DataChunk*>(d);
122
123   for( int i = 0; i < ChunkSize; ++i )
124     fFile << data->a[i] << endl;
125
126 #ifndef MUTE
127    cout << " ChunkStored " << endl;
128 #endif
129   return 0;
130 }
```

Typical output after solution

```
 Time:    404.545074 ms.
```

The obtained speed up factor is 2. The small speed up factor is due to dependence of all threads, which makes data saving into the Input/Output device. Note, that writing to the disk is the most time-consuming operation.

HPC Practical Course
Part 4.1

Open Computing Language (OpenCL)

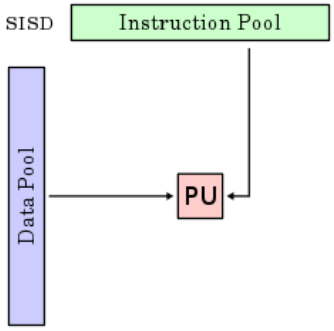
V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak

Goethe University of Frankfurt am Main

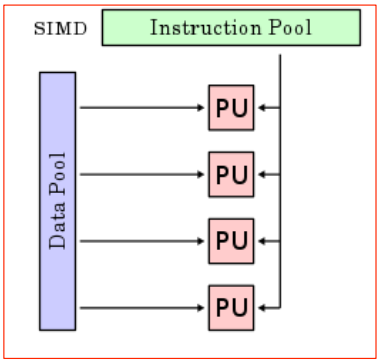
11 June 2014

Computer Architectures

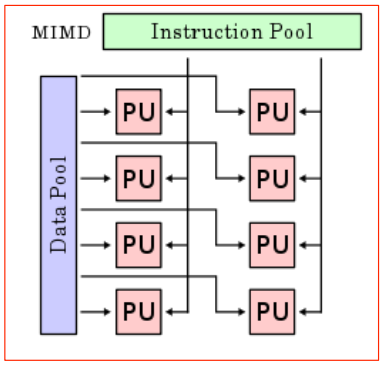
Single Instruction Single Data



Single Instruction Multiple Data



Multiple Instruction Multiple Data

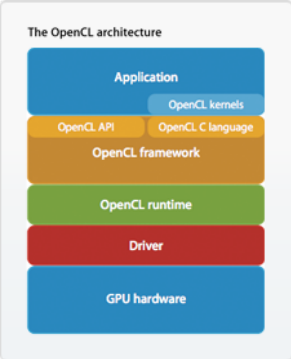


Taken from: http://en.wikipedia.org/wiki/Flynn's_taxonomy

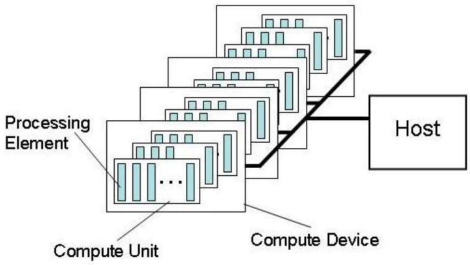
OpenCL Architecture

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors.

- **Platform Layer API**
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialise compute devices
  - Create compute contexts and work-queues
- **Runtime API**
  - Execute compute kernels
  - Manage scheduling, compute, and memory resources
- **Language Specification**
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions - familiar to developers
  - Defined numerical accuracy - IEEE 754 rounding with specified maximum error
  - Online or offline compilation and build of compute kernel executables
  - Rich set of built-in functions
- **Practicality, flexibility and retargetability**



Programming Model



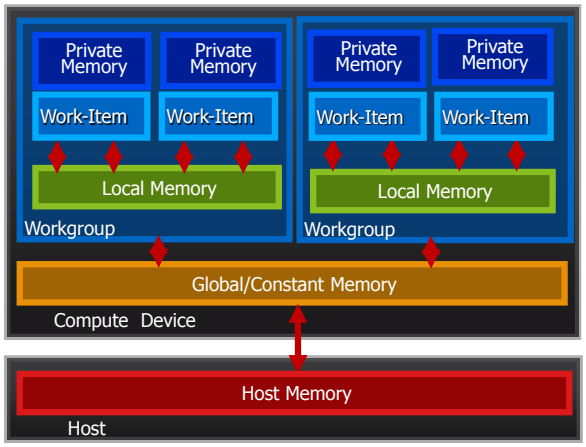
- **Host**
  - run the main program
  - run the compilation
  - distributes tasks between compute devices
  - tasks are distributed via queues
- **Compute devises**
  - example - CPU or GPU
  - consists of one or more Compute units
- **Compute units**
  - example - set of cores of CPU, streaming multiprocessor of GPU
  - consists of one more Processing elements
- **Processing elements**
  - example - one core of CPU, one core of GPU

Memory Model

global memory space - largest memory space available to the device.

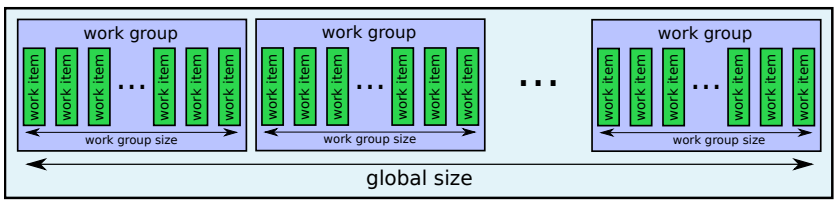
Each compute unit on the device has a local memory, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any work-item in a work-group, and thus allows for local communication between work-groups.

Additionally, attached to each processing element is a private memory, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing element's registers.

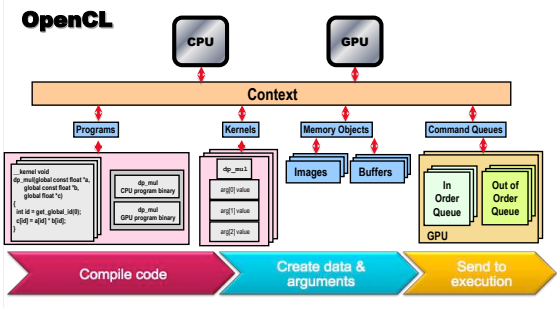


Execution Model

- **OpenCL application runs on a host which submits work to the compute devices**
  - Context: The environment within which work-items executes, includes devices and their memories and command queues
  - Program: Collection of kernels and other functions (Analogous to a dynamic library)
  - Kernel: the code for a work item. Basically a C function
  - Work item: the basic unit of work on an OpenCL device
  - Each processing element works on one work item
  - Work items are combined into working group
  - Each working group is assigned to the compute unit
- **Applications queue kernel execution**
  - Executed in-order or out-of-order



Host-program structure



- Get a platform
  - Get a device
    - Set a context
      - Create a command-queue
        - Create memory buffer
        - Write the buffer
          - Create a program
          - Compile the program
            - Create a kernel
            - Set the kernel arguments
            - Call the kernel
        - Read the buffer
- Clean the memory

Host Program Structure

Platform

```
// Returns the error code
cl_int oclGetPlatformID (cl_platform_id *platforms)   // Pointer to the platform object
```

Device

```
// Returns the error code
cl_int clGetDeviceIDs  (cl_platform_id platform,
          cl_device_type device_type,   // Bitfield identifying the type. For the GPU
we use CL_DEVICE_TYPE_GPU
          cl_uint num_entries,   // Number of devices, typically 1
          cl_device_id *devices,   // Pointer to the device object
          cl_uint *num_devices)   // Puts here the number of devices matching the
device_type
```

Context

```
// Returns the context
cl_context clCreateContext (const cl_context_properties *properties,   // Bitwise with
the properties (see specification)
                cl_uint num_devices,   // Number of devices
                const cl_device_id *devices,   // Pointer to the devices object
                void (*pfn_notify)(const char *errinfo, const void *private_info,
size_t cb, void *user_data),   // (don't worry about this)
                void *user_data,   // (don't worry about this)
                cl_int *errcode_ret)   // error code result
```

http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201

Host Program Structure

Command queue

```
cl_command_queue clCreateCommandQueue (cl_context context,
                cl_device_id device,
                cl_command_queue_properties properties,   // Bitwise with the
properties
                cl_int *errcode_ret)   // error code result
```

Buffer: create

```
// Returns the cl_mem object referencing the memory allocated on the device
cl_mem clCreateBuffer (cl_context context,   // The context where the memory will be
allocated
          cl_mem_flags flags,
          size_t size,   // The size in bytes
          void *host_ptr,
          cl_int *errcode_ret)
```

- CL_MEM_READ_WRITE
- CL_MEM_WRITE_ONLY
- CL_MEM_READ_ONLY
- CL_MEM_USE_HOST_PTR
- CL_MEM_ALLOC_HOST_PTR
- CL_MEM_COPY_HOST_PTR

Buffer: write

```
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t cb,
                             const void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```


Host Program Structure

Program: create

```
// Returns the OpenCL program
cl_program clCreateProgramWithSource (cl_context context,
                 cl_uint count,   // number of files
                 const char **strings,   // array of strings, each one is a file
                 const size_t *lengths,   // array specifying the file lengths
                 cl_int *errcode_ret)   // error code to be returned
```

Program: build

```
cl_int clBuildProgram (cl_program program,
                 cl_uint num_devices,
                 const cl_device_id *device_list,
                 const char *options,   // Compiler options, see the specifications
for more details
                 void (*pfn_notify)(cl_program, void *user_data),
                 void *user_data)
```

Error log:

```
cl_int clGetProgramBuildInfo (cl_program program,
                cl_device_id device,
                cl_program_build_info param_name,   // The parameter we want to know
                size_t param_value_size,
                void *param_value,    // The answer
                size_t *param_value_size_ret)
```

- CL_PROGRAM_BUILD_STATUS
- CL_PROGRAM_BUILD_OPTIONS
- CL_PROGRAM_BUILD_LOG

Kernel: create

```
cl_kernel clCreateKernel (cl_program program,   // The program where the kernel is
        const char *kernel_name,   // The name of the kernel, i.e. the name of the
kernel function as it's declared in the code
        cl_int *errcode_ret)
```

Host Program Structure

Kernel: arguments

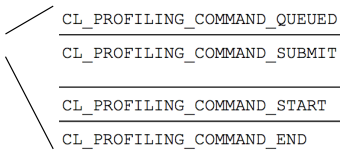
```
cl_int  clSetKernelArg (cl_kernel kernel,   // Which kernel
          cl_uint arg_index,   // Which argument
          size_t arg_size,   // Size of the next argument (not of the value pointed by it!)
          const void *arg_value)    // Value
```

Kernel: call

```
cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                            cl_kernel kernel,
                            cl_uint  work_dim,    // Choose if we are using 1D, 2D or 3D work-items and work-groups
                            const size_t *global_work_offset,
                            const size_t *global_work_size,   // The total number of work-items (must have work_dim dimensions)
                            const size_t *local_work_size,     // The number of work-items per work-group (must have work_dim dimensions)
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

Profile

```
cl_int clGetEventProfilingInfo (cl_event event,
                                cl_profiling_info param_name,
                                size_t param_value_size,
                                void *param_value,
                                size_t *param_value_size_ret)
```



Host Program Structure

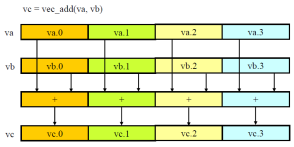
Buffer: read

```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                        cl_mem buffer,   // from which buffer
                        cl_bool blocking_read,   // whether is a blocking or non-blocking read
                        size_t offset,   // offset from the beginning
                        size_t cb,   // size to be read (in bytes)
                        void *ptr,   // pointer to the host memory
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event)
```

Clean:

```
// Cleaning up
delete[] src_a_h;
delete[] src_b_h;
delete[] res_h;
delete[] check;
clReleaseKernel(vector_add_k);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(src_a_d);
clReleaseMemObject(src_b_d);
clReleaseMemObject(res_d);
```

Vectorization with OpenCL



Vc	OpenCL
int_v	int4
float_v	float4
store()	vstore4()
load()	vload4()
...	...

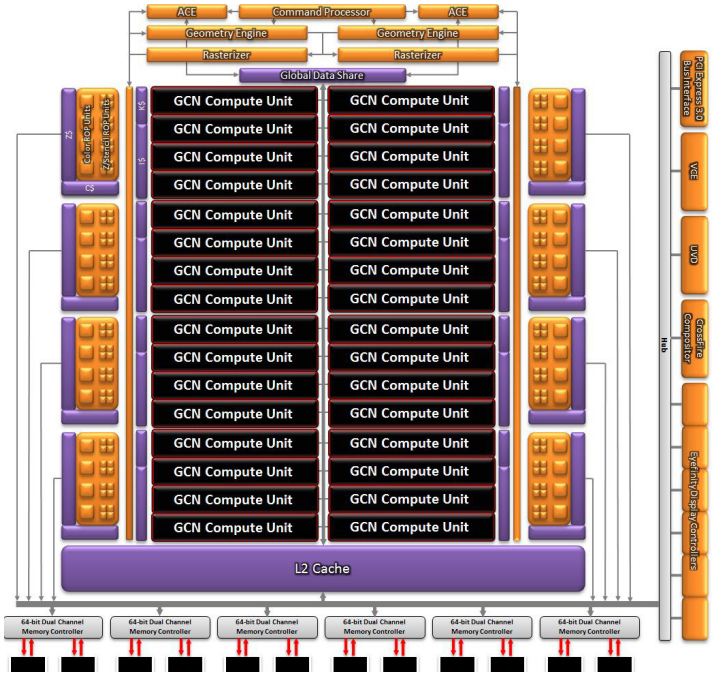
Example: increase 12,13,14 and 15-th elements of array A by one

```
int A[1000];
int4 a = vload4( 3, A );
a++;
vstore4( a, 3, A );
```


GPU Memory

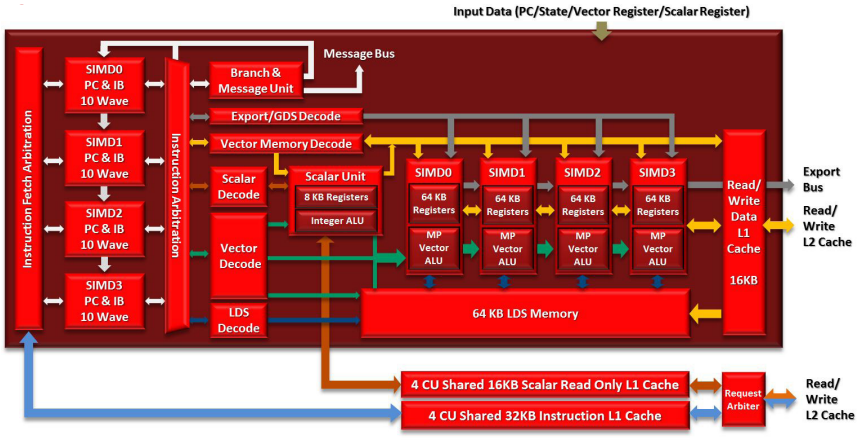
- **Global memory**
  - very large, typically gigabytes
  - readable and writable by all work items
  - state only well defined after kernel has finished
  - slow, but much faster with streaming access (coalescing)
  - sometimes cached
- **Constant memory**
  - read-only part of the global memory (writable from host)
  - often cached
  - prefer constant memory for constant values
- **Local memory**
  - very fast on-chip memory
  - shared among work items within the same work group
  - versatile! (explicit global memory cache, etc.)
- **Private memory**
  - private to a single work item
  - usually physically a part of the global memory, slow!
  - will be used to store the work items registers if the register file is exhausted (must be avoided!)

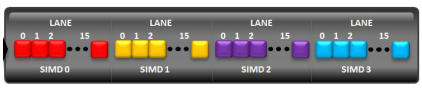
Structure of AMD Radeon HD 7970



- 32 compute units
- Up to 925MHz Engine Clock
- 3GB GDDR5 Memory
- 3.79 TFLOPS Single Precision compute power

GCN Compute Unit





- 16 floats per SIMD lane
- 64 floats in total per Compute Unit

Exercise 0

An example code is given. It computes vector sum C = A + B.

1. Part 1
    1. Run and understand
    2. Check error codes, returned by each function (they should be equal to CL_SUCCESS==0)
    3. Play: try to change size of the arrays (try 128, 64, 16, 1023), type (try float), etc.
    4. Solution is main1.cpp
2. Part 2
    1. Display build log
    2. Measure the execution time
        1. for comparison implement scalar version
        2. try more complicated computations (log, sqrt)
    3. Solution is main2.cpp
3. Part 3: SIMDize
    1. Solution is main3.cpp

Exercise 0 (continue)

4. Part 4: Create sub devices
    1. Create sub devices with

```
cl_int clCreateSubDevices ( cl_device_id in_device ,
                const cl_device_partition_property  *properties ,
                cl_uint  num_devices ,
                cl_device_id  *out_devices ,
                cl_uint  *num_devices_ret )
```

    2. Try CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS and CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN properties (more information you can find here: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html). Solution is main4.cpp
5. Part 5:
    1. Create a function into the kernel function for a sum calculation
    2. We suggest to build the program in c++-like style: clBuildProgram(program, 1, &out_devices[0], "-x clc++", NULL, NULL);
    3. Solution is main5.cpp

Exercise 0 (continue)

6. Part 6: Run on GPU
    1. Try SIMD and scalar versions
    2. Try different sizes of working groups
    3. Solution is main6.cpp

Exercise 1. SIMD KF

- Implement SIMD KF package with OpenCL:
    - Implement the host part in Fit.cxx (find TODO)
    - Finish the Fit.cl file: implement kernel function, describe data structures. Functionality is already there
- Measure the scalability with OpenCL:
    - cd TimeHisto
    - . ~/pandaroot/trunk/build/config.sh
    - root -l make_timehisto_stat_complex_opencl.C

4.1. OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. It allows to write the universal code, which can be run both on the CPU and GPU giving software developers portable and efficient access to the power of the heterogeneous processing platforms. OpenCL supports a wide range of applications through a low-level, high-performance, portable abstraction. OpenCL consists of an API for coordinating parallel computation across heterogeneous processors and a cross-platform programming language with a well-specified computation environment. The OpenCL standard:

- supports both data- and task-based parallel programming models;
- utilises a subset of ISO C99 with extensions for parallelism;
- defines consistent numerical requirements based on IEEE 754;
- defines a configuration profile for handheld and embedded devices Efficiently interoperates with OpenGL, OpenGL ES and other graphics APIs.

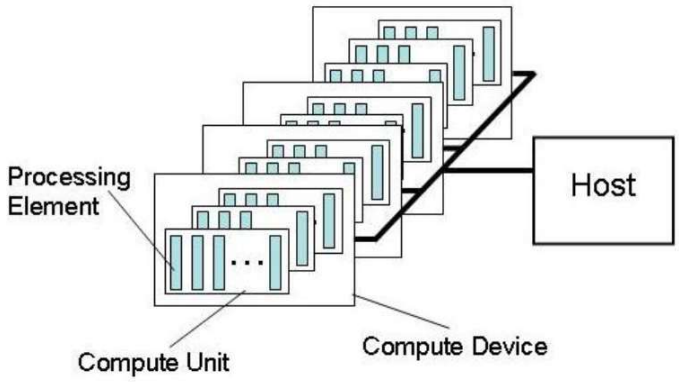


Fig. 1. Programming model of OpenCL.

OpenCL includes runtime API, which compiles kernels, Manage scheduling, compute, and memory resources and executes kernels.

The programming model of OpenCL (see Fig. 1) consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (for example, one CPU of the server or one streaming multiprocessor of the GPU), which are further divided into one or more processing elements (cores of the CPU or streaming multiprocessor). The OpenCL application submits commands from the host to execute computations on the processing elements within a device. An OpenCL application runs on a host according to the models native to the host platform. The processing elements within a compute unit execute a single stream of instructions as SIMD units or each processing element maintains its own program counter.

The high abstraction level requires to specify the memory model. The memory in OpenCL is divided into several layers (Fig. 2). The host and device memory are separated. The largest memory available to the device is called a **global memory**. For CPU, for example, the global memory is RAM of the server, for GPU - RAM of the GPU. Usually, global memory is the slowest one. Some part of the global memory is considered as a **constant memory**. It is usually faster then global, because of caching. Each compute unit on the device has a **local memory**, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any work-item in a work-group, and thus allows for local communication between. Additionally, attached to each processing element is a **private memory**, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing element's registers. Usually the private memory physically is a part of the global, therefore it is also slow.

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution. Also it defines a queue of the tasks and runs corresponding kernels according to the queue. The data is divided into work groups (Fig. 3), which are assigned to the compute

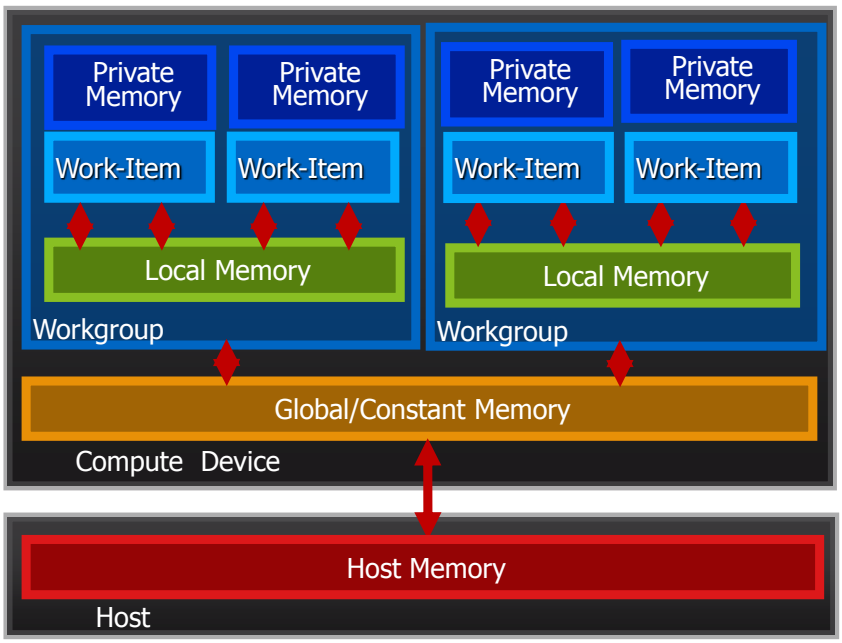


Fig. 2. The memory model of OpenCL.

unit. Each work group consist of work items. Work item is a basic unit of work, it runs the instance of the kernel on the individual processing element.

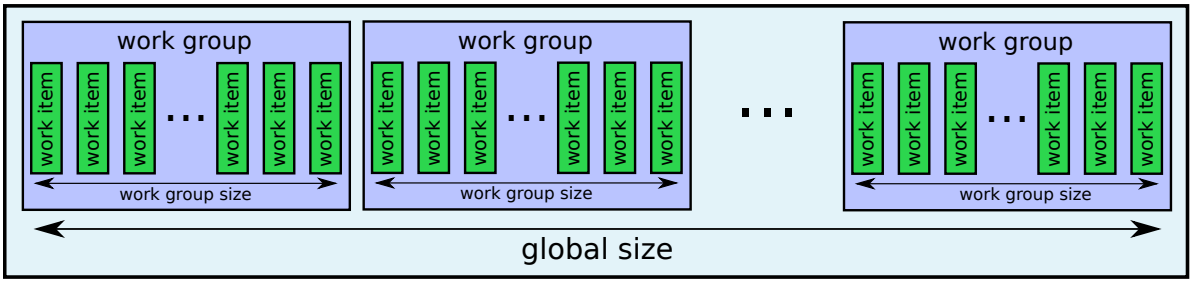


Fig. 3. Organisation of the data in OpenCL.

The structure of the host program:

- get a platform - the information about the whole;
- get a device - select the device for computations;
- set a context within which the program will work;
- create a command-queue;
- create a memory buffer;
- write the buffer (fill the buffer with the input data);
- create a program - an OpenCL object, the input for it - a *.cl file with the main kernel function;
- compile the program;
- create a kernel;
- set the kernel arguments;
- call the kernel;
- read the buffer;
- clean the memory.

Let us describe the functionality of OpenCL used for this. More detailed description can be found here: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.

1) cl_int clGetPlatformIDs(cl_uint num_entries,
cl_platform_id *platforms,
cl_uint *num_platforms)

Obtain the list of platforms available. num_entries - the maximum number of elements in platforms array, platforms - returned array of platform ids, num_platforms - returns number of available platforms. The function returns the error code.

2) cl_int clGetDeviceIDs (cl_platform_id platform ,
cl_device_type device_type ,
cl_uint num_entries ,
cl_device_id *devices ,
cl_uint *num_devices)

Obtain the list of devices available on a platform. platform - id of the platform, where we are looking for a device, device_type - type of a device (CL_DEVICE_TYPE_CPU to use CPU, CL_DEVICE_TYPE_GPU to use GPU or CL_DEVICE_TYPE_ALL to use both of them), num_entries - the maximum number of elements in devices array, devices - returned array of devices ids, num_devices - returns number of OpenCL devices available that match device_type. The function returns the error code.

3) cl_context clCreateContext(const cl_context_properties *properties,
cl_uint num_devices,
const cl_device_id *devices,
(void CL_CALLBACK *pfn_notify) (const char *errinfo,
const void *private_info, size_t cb,
void *user_data),
void *user_data,
cl_int *errcode_ret)

Creates an OpenCL context. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. properties - specifies a list of context property names and their corresponding values, num_devices - the number of devices specified in the devices argument; devices - array with device ids, which will be used within current context; errcode_ret - the error code returned by the function.

4) cl_command_queue clCreateCommandQueue(cl_context context,
cl_device_id device,
cl_command_queue_properties properties,
cl_int *errcode_ret)

Create a command-queue on a specific device. context - valid OpenCL context created before; device - id of a device from the context, for which the queue is created; properties - properties of the queue, if profiling should be enabled the value should be CL_QUEUE_PROFILING_ENABLE, if the execution mode should be out of order the value should be CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE; errcode_ret - the error code returned by the function.

5) cl_mem clCreateBuffer (cl_context context,
cl_mem_flags flags,
size_t size,
void *host_ptr,
cl_int *errcode_ret)

Creates a buffer object. context - valid OpenCL context created before; flags - specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used, value can be, for example, CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY; size - the size in bytes of the buffer memory object to be allocated; host_ptr - a pointer to the buffer data that may already be allocated by the application, the size of the buffer that host_ptr points to must be ≥ size bytes; errcode_ret - the error code returned by the function.

6) cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_write,
size_t offset,
size_t size,
const void *ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueue commands to write to a buffer object from host memory. command_queue - refers to the command-queue in which the write command will be queued, command_queue and buffer must be created with the same OpenCL context; buffer - refers to a valid buffer object, offset - the offset in bytes in the buffer object to write to, size - the size in bytes of data being written; ptr - the pointer to buffer in host memory where data is to be written from; event_wait_list, num_events_in_wait_list - array of events together with its size to be waited before execution of the current function; event - returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete. The function returns the error code.

7) cl_program clCreateProgramWithSource (cl_context context,
cl_uint count,
const char **strings,
const size_t *lengths,
cl_int *errcode_ret)

Creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object. context - valid OpenCL context created before; count - number of files (strings) to be compiled; strings - array of strings containing the source code; lengths - array with sizes of each string; errcode_ret - the error code returned by the function.

8) cl_int clBuildProgram (cl_program program,
cl_uint num_devices,
const cl_device_id *device_list,
const char *options,
void (CL_CALLBACK *pfn_notify)(cl_program program, void
*user_data),
void *user_data)

Builds (compiles and links) a program executable from the program source or binary. program - the program object; device_list - a pointer to a list of devices associated with the program; num_devices - the number of devices listed in device_list; options - compilation options, to build the program supporting c++-like functionality should be "-x clc++". The program returns the error code.

9) cl_int clGetProgramBuildInfo (cl_program program,
cl_device_id device,
cl_program_build_info param_name,
size_t param_value_size,
void *param_value,
size_t *param_value_size_ret)

Returns build information for each device in the program object. We will use it to print the build log. program - the program object being queried; device - specifies the device for which build information is being queried; param_name - specifies the information to query, in our case should be CL_PROGRAM_BUILD_LOG; param_value_size - the size in bytes of memory pointed to by param_value; param_value - a pointer to memory where the appropriate result being queried is returned; param_value_size_ret - returns the actual size of the log. The function returns the error code.

10) cl_kernel clCreateKernel (cl_program program,
const char *kernel_name,
cl_int *errcode_ret)

Creates a kernel object. program - a program object with a successfully built executable, kernel_name - a function name in the program declared with the __kernel qualifier; errcode_ret - the error code returned by the function.

11) cl_int clSetKernelArg (cl_kernel kernel,
cl_uint arg_index,
size_t arg_size,
const void *arg_value)

Used to set the argument value for a specific argument of a kernel. kernel - a valid kernel object; arg_index - the argument index, starts from 0; arg_size - specifies the size of the argument value; arg_value - a pointer to data that should be used as the argument value for argument specified by arg_index. The function returns the error code.

12) cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
cl_kernel kernel,
cl_uint work_dim,
const size_t *global_work_offset,
const size_t *global_work_size,
const size_t *local_work_size,

cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueues a command to execute a kernel on a device. command_queue - a valid command-queue, the kernel will be queued for execution on the device associated with command_queue; kernel - a valid kernel object; work_dim - the number of dimensions used to specify the global work-items and work-items in the work-group; global_work_offset - can be used to specify an array of work_dim unsigned values that describe the offset used to calculate the global ID of a work-item; global_work_size - points to an array of work_dim unsigned values that describe the number of global work-items in work_dim dimensions that will execute the kernel function; local_work_size - points to an array of work_dim unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by kernel; event_wait_list, num_events_in_wait_list - array of events together with its size to be waited before execution of the current function; event - returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete.

13) cl_int clGetEventProfilingInfo (cl_event event,
cl_profiling_info param_name,
size_t param_value_size,
void *param_value,
size_t *param_value_size_ret)

Returns profiling information for the command associated with event if profiling is enabled. event - event to be profiled; param_name - specifies the profiling data to query (CL_PROFILING_COMMAND_QUEUED, CL_PROFILING_COMMAND_SUBMIT, CL_PROFILING_COMMAND_START, CL_PROFILING_COMMAND_END); param_value_size - specifies the size in bytes of memory pointed to by param_value; param_value - A pointer to memory where the appropriate result being queried is returned; param_value_size_ret - returns the actual size in bytes of data copied to param_value. The function returns the error code.

14) cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_read,
size_t offset,
size_t size,
void *ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueue commands to read from a buffer object to the host memory. Parameters has the same description as for clEnqueueWriteBuffer().

15) cl_int clCreateSubDevices (cl_device_id in_device ,
const cl_device_partition_property *properties ,
cl_uint num_devices ,
cl_device_id *out_devices ,
cl_uint *num_devices_ret)

Creates an array of sub-devices that each reference a non-intersecting set of compute units within in_device. in_device - the device to be partitioned; properties - specifies how in_device should be partitioned described by a partition name and its corresponding value (CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS, CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN); num_devices - size of memory pointed to by out_devices specified as the number of cl_device_id entries; out_devices - the buffer where the OpenCL sub-devices will be returned; num_devices_ret - returns the number of sub-devices that device may be partitioned into according to the partitioning scheme specified in properties.

7_OpenCL/1_First: description

The first exercise is a simple program, which computes vector sum C=A+B. It consist of two parts: the host program (**main.cpp**) and the OpenCL kernel (**vector_add_kernel.cl**). The tasks for this exercise are:

Part 1:

- run and understand the code;
- check error codes, returned by each function (they should be equal to CL_SUCCESS==0);

- play: try to change size of the arrays (try 128, 64, 16, 1023), type (try float), etc.;
- solution is **main1.cpp** and **vector_add_kernel.cl**.

Part 2:

- display build log;
- measure the execution time
    1. increase the size of the array to 1000000, increase the **local_item_size**;
    2. because of the increased time comment the printing of the result on the screen;
    3. for comparison implement scalar version;
    4. try more complicated computations (log, sqrt);
- solution is **main2.cpp** and **vector_add_kernel.cl**.

Part 3:

- SIMDize;
- Solution is **main3.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

Part 4:

- create sub devices;
- try CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS and CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN properties (more information you can find here: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html);
- solution is **main4.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

Part 5:

- create a function into the kernel function for a sum calculation;
- we suggest to build the program in c++-like style: clBuildProgram(program, 1, &out_devices[0], "-x clc++", NULL, NULL);
- solution is **main5.cpp** and **vector_add_kernel.cl4**.

Part 6:

- run on GPU;
- try SIMD and scalar versions;
- try different sizes of working groups;
- solution is **main6.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

7_OpenCL/1_First: solution

Part 1.
To check the error codes we suggest to create a function:

Part of the source code of Solution/main1.cpp

```
22 inline void
23 checkErr(cl_int err, const char * name)
24 {
25     if (err != CL_SUCCESS) {
26         std::cerr << "ERROR: " << name
27                   << " (" << err << ")" << std::endl;
28         exit(EXIT_FAILURE);
29     }
30 }
```

and call it after each OpenCL function, for example, after getting the platform:

Part of the source code of Solution/main1.cpp

```
67 checkErr(ret, "clGetPlatformIDs");
```

To change the type throughout the whole code easily we introduced a type **DataType**:

Part of the source code of Solution/main1.cpp

```
32 typedef float DataType;
```

It can be set to int or float. The code of the program should be modified respectively:

Part of the source code of Solution/main1.cpp

```
38      DataType *A = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
39      DataType *B = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
…
80      cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
81              LIST_SIZE * sizeof(DataType), NULL, &ret);
82      cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
83              LIST_SIZE * sizeof(DataType), NULL, &ret);
84      cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
85              LIST_SIZE * sizeof(DataType), NULL, &ret);
…
88      ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
89              LIST_SIZE * sizeof(DataType), A, 0, NULL, NULL);
90      checkErr(ret, "clEnqueueWriteBuffer");
91      ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
92              LIST_SIZE * sizeof(DataType), B, 0, NULL, NULL);
…
120     DataType *C = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
121     ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
122             LIST_SIZE * sizeof(DataType), C, 0, NULL, NULL);
```

and the OpenCL kernel:

Part of the source code of Solution/main1.cpp

```
1   __kernel void vector_add(__global float *A, __global float *B, __global float
    *C) {
2
3       // Get the index of the current element
4       int i = get_global_id(0);
5
6       // Do the operation
7       C[i] = A[i] + B[i];
8   }
```

When changing the size of the arrays to 128 and 64 the program works fine. When changing it to 16 or 1023 - the OpenCL program will not be compiled, because the size of the array should be dividable on **local_item_size**. Setting **local_item_size**, for example, to one will solve the problem.

Part 2.

To extract the build log next lines should be added:

Part of the source code of Solution/main2.cpp

```
114     // Shows the log
115     char* build_log;
116     size_t log_size;
117     // First call to know the proper size
118     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 0, NULL,
    &log_size);
119     build_log = new char[log_size+1];
120     // Second call to get the log
121     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, log_size,
    build_log, NULL);
122     build_log[log_size] = '\0';
123     cout << build_log << endl;
124     delete[] build_log;
```

If we will underestimate the size, the log will be incomplete, if we will overestimate it - the log will be displayed together with a large empty region. Therefore the first time clGetProgramBuildInfo is called to calculate the size of the log. And the second time we get the log itself to the variable **build_log**.

To compare times of the OpenCL code and a normal c++ code the scalar function should be implemented:

Part of the source code of Solution/main2.cpp

```
37 void scalar_add( DataType A,  DataType B,  DataType C) {
38
39       // Do the operation
40     C = A + B;
41 }
```

and called together with the time measurement:

Part of the source code of Solution/main2.cpp

```
164     TStopwatch timer;
165     for(i = 0; i < LIST_SIZE; i++)
166        scalar_add( A[i], B[i], C[i] );
167     timer.Stop();
```

To improve the precision of the time measurement the size of the array should be increased:

Part of the source code of Solution/main2.cpp

```
47     const int LIST_SIZE = 1000000;
```

To enable profiling of the OpenCL code the queue parameters should be modified:

Part of the source code of Solution/main2.cpp

```
88       cl_command_queue command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
```

the event should be generated on the kernel execution and we should wait for this event:

Part of the source code of Solution/main2.cpp

```
138     cl_event event;
139
140     ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
141             &global_item_size, &local_item_size, 0, NULL, &event);
142     checkErr(ret, "clEnqueueNDRangeKernel");
143
144     ret = clWaitForEvents(1 , &event);
```

And when the event is finished we can profile it and print the time:

Part of the source code of Solution/main2.cpp

```
153     cl_ulong time_start, time_end;
154     double total_time;
155     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(time_start), &time_start, NULL);
156     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end),
&time_end, NULL);
157     total_time = time_end - time_start;
158     printf("Parallel time = %0.3f ms\n", (total_time / 1000000.0) );
```

To improve the performance and decrease overhead the **local_item_size** should be increased. But even after this the OpenCL code is mush slower. With more complicated calculations this will change.

Part 3.

To have the SIMD and scalar code in the same file we introduced a preprocessor macro:

Part of the source code of Solution/main3.cpp

```
9  #define SIMD // switch between vectorized and not vectorized versions
```

Here we again decrease the size of the array:

Part of the source code of Solution/main3.cpp

```
49      const int LIST_SIZE = 1024;
```

Also we need to introduce a new kernel (**vector_add_kernel2.cl**):

Part of the source code of Solution/vector_add_kernel2.cl

```
1  __kernel void vector_add(__global float *A, __global float *B, __global float
   *C) {
2
3      // Get the index of the current element
4      int i = get_global_id(0);
5
6      // get the i-th group of 4
7      float4 a = vload4(i, A);
8      float4 b = vload4(i, B);
9
10     // store a+b to 4*i-th element
11     vstore4( a + b, i, C );
12 }
```

And we need to load corresponding kernel:

Part of the source code of Solution/main3.cpp

```
62 #ifdef SIMD
63     fp = fopen("vector_add_kernel2.cl", "r");
64 #else
65     fp = fopen("vector_add_kernel.cl", "r");
66 #endif
```

and modify **global_item_size**:

Part of the source code of Solution/main3.cpp

```
62 #ifdef SIMD
63     size_t global_item_size = LIST_SIZE/4; // Process the entire lists
64 #else
65     size_t global_item_size = LIST_SIZE;
66 #endif
```

Part 4.

To create subdevices the code should be added:

Part of the source code of Solution/main4.cpp

```
89      cl_uint num_devices_ret;
90      cl_device_id  out_devices[80];
91      /// CL_DEVICE_PARTITION_EQUALLY
```

Part of the source code of Solution/main4.cpp

```
92      const cl_device_partition_property props[] = {CL_DEVICE_PARTITION_EQUALLY,
        2, 0};
93      ret = clCreateSubDevices ( device_id, props, 80 , out_devices ,
        &num_devices_ret );
94      checkErr(ret, "clCreateSubDevices");
95      /// CL_DEVICE_PARTITION_BY_COUNTS
96  //      const cl_device_partition_property props[] =
        {CL_DEVICE_PARTITION_BY_COUNTS, 1, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END,
        0};
97  //      ret = clCreateSubDevices ( device_id, props, 80 , out_devices ,
        &num_devices_ret );
98  //      checkErr(ret, "clCreateSubDevices");
99      ///CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
100 //      const cl_device_partition_property props[] =
        {CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE,
        0};
101 //      ret = clCreateSubDevices ( device_id, props, 80 , out_devices ,
        &num_devices_ret );
102 //      checkErr(ret, "clCreateSubDevices");
```

and we will use only the first device:

Part of the source code of Solution/main4.cpp

```
109          cl_command_queue  command_queue  =  clCreateCommandQueue(context,
        out_devices[0], CL_QUEUE_PROFILING_ENABLE, &ret);
128     cl_program program = clCreateProgramWithSource(context, 1,
129             (const char **)&source_str, (const size_t *)&source_size, &ret);
139        clGetProgramBuildInfo(program,  out_devices[0],  CL_PROGRAM_BUILD_LOG,  0,
        NULL, &log_size);

142     clGetProgramBuildInfo(program, out_devices[0], CL_PROGRAM_BUILD_LOG,
        log_size, build_log, NULL);
```

Part 5.

In this exercise we will work only with a SIMD version. We should load the corresponding file with a kernel:

Part of the source code of Solution/main5.cpp

```
60      fp = fopen("vector_add_kernel4.cl", "r");
```

should build the kernel with c++-option:

Part of the source code of Solution/main5.cpp

```
127     ret = clBuildProgram(program, 1, &out_devices[0], "-x clc++", NULL, NULL);
```

And the function should be added to the kernel:

Part of the source code of Solution/vector_add_kernel4.cl

```
1   void Add(float4 &a, float4 &b, float4 &sum)
2   {
3     sum = a+b;
4   }
5
6   __kernel void vector_add(__global float *A, __global float *B, __global float
    *C) {
7
8       // Get the index of the current element
9       int i = get_global_id(0);
```

Part of the source code of Solution/vector_add_kernel4.cl

```
10
11      // get the i-th group of 4
12      float4 a = vload4(i, A);
13      float4 b = vload4(i, B);
14
15      float4 sum;
16      Add(a,b,sum);
17
18      // store a+b to 4*i-th element
19      vstore4( sum, i, C );
20  }
```

7_OpenCL/2_SIMDKF: description

In the second exercise we will implement a more complex program with OpenCL. This program is SIMD KF track fitter for the CBM experiment, which we already implemented with Vc, header files and OpenMP. The program we will be run on CPU.

At first, the main file **Fit.cxx**, which call the fitting function, should be modified: the OpenCL part should be added here to manage the devices and to distribute tracks between cores. This part should be added at lines:

Part of the source code of Fit.cxx

```
528     //TODO write your code here: get a platform, a CPU device, create
        subdevices (use CL_DEVICE_PARTITION_EQUALLY),
529     //create a context, a command queue, memory buffers on the device, copy
        data to the buffers,
530     //create a program from the kernel source, build the program, check the
        logs, create a kernel, set its arguments,
531     //execute the kernel, read the fitted tracks, clean up the memory
```

The program should get the platform, then get a CPU device and create sub devices in order to be able to measure the scalability. Then a context should be declared and within this context a command queue and memory buffers should be created and the buffers should be filled with data. Then a program from the **Fit.cl** file should be created and compiled, the log should be checked for the correctness of compilation, the kernel should be created and initialised with arguments. When the kernel is ready it should be queued and executed on the device. For profiling the event on kernel execution should be generated. When the OpenCL program would finish, read the fitted tracks back to the host program and clean the memory.

The file with OpenCL kernel is **Fit.cl**. It is already partially created: the fitting functionality is described there. The first task will be to describe there data structures used by the functions (see line 7): **FieldVector**, **FieldSlice**, **FieldRegion**, **Station**, **HitV**, **CovV** and **TrackV**. The data structures should be identical to those described in the **FitClasses.h** file. The second task is to write the kernel, which runs **Fit()** function (see line 542).

To compile the program type **make opencl**. To run - type **./opencl n**, where **n** is a number of threads to be used.

When the program will be ready, check that it works correctly:

- run the program with 1 thread: **./opencl 1**;
- go to the folder with QA macro: **cd QualityHisto/**;
- plot the fit QA histograms: **root -l -b -q histo_particle.C; root -l Pulls.C**;
- compare obtained **KFTrackPull.pdf** with the **KFTrackPull_etalon.pdf**, their should be almost the same, minor changes are negligible.

After this go to the folder **TimeHisto** and measure the scalability:

- run bash script to collect the data: **. make_data_opencl.sh** ;
- plot the scalability: **root -l make_timehisto_stat_complex_opencl.C** .

7_OpenCL/2_SIMDKF: solution

The host program should be modified as follows. At first, we should get the platform:

Part of the source code of Solution/Fit.cxx

```
529     cl_platform_id platform_id = NULL;
530     cl_device_id device_id = NULL;
531     cl_uint ret_num_devices;
532     cl_uint ret_num_platforms;
533     cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
534 //      ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,
535 //              &device_id, &ret_num_devices);
536     checkErr(ret, "clGetPlatformIDs");
```

Then we should get a CPU device, specifying **CL_DEVICE_TYPE_CPU**:

Part of the source code of Solution/Fit.cxx

```
538     ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_CPU, 1,
539             &device_id, &ret_num_devices);
540     checkErr(ret, "clGetDeviceIDs");
```

Then the devise should be partitioned equally and each partition should contain **tasks** cores, the output devices will be stored to the array **cl_device_id out_devices[80]** :

Part of the source code of Solution/Fit.cxx

```
542     cl_uint num_devices_ret;
543     cl_device_id  out_devices[80];
544     const cl_device_partition_property props[] = {CL_DEVICE_PARTITION_EQUALLY,
    tasks, 0};
545     ret = clCreateSubDevices ( device_id, props, 80 , out_devices ,
    &num_devices_ret );
546     checkErr(ret, "clCreateSubDevices");
```

Then a context with a queue should be created with profiling enabled:

Part of the source code of Solution/Fit.cxx

```
548     // Create an OpenCL context
549     cl_context context = clCreateContext( NULL, 1, &out_devices[0], NULL, NULL,
    &ret);
550
551     // Create a command queue
552     cl_command_queue command_queue = clCreateCommandQueue(context,
    out_devices[0], CL_QUEUE_PROFILING_ENABLE, &ret);
```

Then the buffers for tracks, stations and numbers of track fits and stations should be created and filled:

Part of the source code of Solution/Fit.cxx

```
554     // Create memory buffers on the device for each vector
555     cl_mem track_mem_obj = clCreateBuffer(context, CL_MEM_READ_WRITE,
556             NTracksV * sizeof(TrackV), NULL, &ret);
557     cl_mem station_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
558             NStations * sizeof(Station), NULL, &ret);
559     cl_mem NStations_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
560             sizeof(int), NULL, &ret);
561     cl_mem NFits_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
```

Part of the source code of Solution/Fit.cxx

```
562             sizeof(int), NULL, &ret);
563
564     // Copy tracks and stations to their respective memory buffers
565     ret = clEnqueueWriteBuffer(command_queue, track_mem_obj, CL_TRUE, 0,
566             NTracksV * sizeof(TrackV), TracksV, 0, NULL, NULL);
567     checkErr(ret, "clEnqueueWriteBuffer");
568     ret = clEnqueueWriteBuffer(command_queue, station_mem_obj, CL_TRUE, 0,
569             NStations * sizeof(Station), vStations, 0, NULL, NULL);
570     checkErr(ret, "clEnqueueWriteBuffer");
571     ret = clEnqueueWriteBuffer(command_queue, NStations_mem_obj, CL_TRUE, 0,
572             sizeof(int), &NStations, 0, NULL, NULL);
573     checkErr(ret, "clEnqueueWriteBuffer");
574     ret = clEnqueueWriteBuffer(command_queue, NFits_mem_obj, CL_TRUE, 0,
575             sizeof(int), &NFits, 0, NULL, NULL);
576     checkErr(ret, "clEnqueueWriteBuffer");
```

Then the program should be created and compiled with option “-x clc++”, which enables c++-like functionality, and the log should be checked:

Part of the source code of Solution/Fit.cxx

```
578     // Create a program from the kernel source
579     cl_program program = clCreateProgramWithSource(context, 1,
580             (const char **)&source_str, (const size_t *)&source_size, &ret);
581
582     // Build the program
583     ret = clBuildProgram(program, 1, &out_devices[0], "-x clc++", NULL, NULL);
584     checkErr(ret, "clBuildProgram");
585
586     // Shows the log
587     char* build_log;
588     size_t log_size;
589     // First call to know the proper size
590     clGetProgramBuildInfo(program, out_devices[0], CL_PROGRAM_BUILD_LOG, 0,
NULL, &log_size);
591     build_log = new char[log_size+1];
592     // Second call to get the log
593     clGetProgramBuildInfo(program, out_devices[0], CL_PROGRAM_BUILD_LOG,
log_size, build_log, NULL);
594     build_log[log_size] = '\0';
595     if(log_size > 1)
596       cout << build_log <<  endl;
597     delete[] build_log;
```

Then the kernel should be created and its arguments should be set:

Part of the source code of Solution/Fit.cxx

```
599     // Create the OpenCL kernel
600     cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
601
602     // Set the arguments of the kernel
603     ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&track_mem_obj);
604     ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&station_mem_obj);
605     ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void
*)&NStations_mem_obj);
606     ret = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *)&NFits_mem_obj);
```

Then the kernel should be executed, the event should be generated on execution, when it will be finished - tracks should be read back and event should be profiled:

Part of the source code of Solution/Fit.cxx

```
611     size_t global_item_size = NTracksV; // Process the entire lists
612 //    size_t local_item_size = NCopy/4; // Process in groups of 64
613     size_t local_item_size = 1; // Process in groups of 64
614 
615     cl_event event;
616 
617     ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
618             &global_item_size, &local_item_size, 0, NULL, &event);
619     checkErr(ret, "clEnqueueNDRangeKernel");
620 
621     ret = clWaitForEvents(1 , &event);
622     checkErr(ret, "clWaitForEvents");
623 
624     // Read the memory buffer track_mem_obj on the device to the local variable
    TracksV
625     ret = clEnqueueReadBuffer(command_queue, track_mem_obj, CL_TRUE, 0,
626             NTracksV * sizeof(TrackV), TracksV, 0, NULL, NULL);
627     checkErr(ret, "clEnqueueReadBuffer");
628 
629     cl_ulong time_start, time_end;
630     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
    sizeof(time_start), &time_start, NULL);
631     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end),
    &time_end, NULL);
632 
633     total_time += (time_end - time_start) / (1.e3 * (double)NTracks *
    (double)NFits);
```

In the end the memory should be cleaned:

Part of the source code of Solution/Fit.cxx

```
636     // Clean up
637     ret = clFlush(command_queue);
638     ret = clFinish(command_queue);
639     ret = clReleaseKernel(kernel);
640     ret = clReleaseProgram(program);
641     ret = clReleaseMemObject(track_mem_obj);
642     ret = clReleaseMemObject(station_mem_obj);
643     ret = clReleaseMemObject(NStations_mem_obj);
644     ret = clReleaseMemObject(NFits_mem_obj);
645     ret = clReleaseCommandQueue(command_queue);
646     ret = clReleaseContext(context);
```

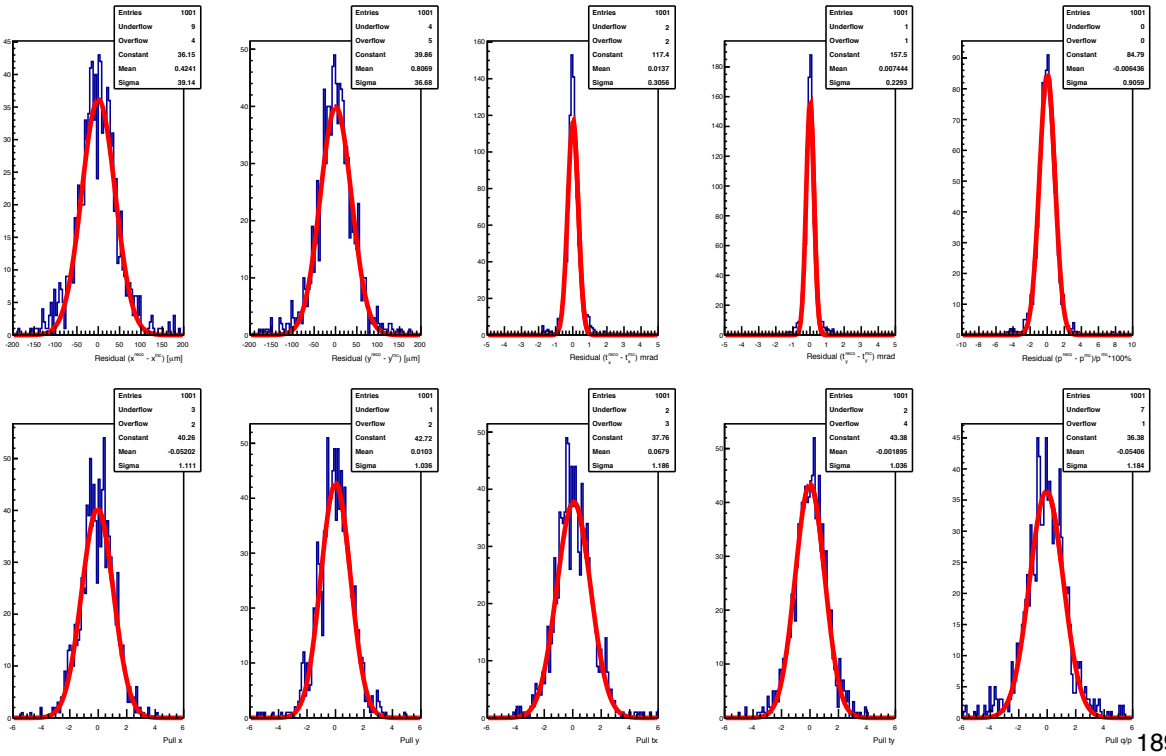
In the **Fit.cl** file the classes should be described. They can be copied from FitClasses.h replacing **Fvec_t** with **float4**. And the kernel should be added:

Part of the source code of Solution/Fit.cxx

```
701 __kernel void vector_add( __global TrackV *t,
702                            __global Station *vStations,
703                            __global int *NStations,
704                            __global int *NFits)
705 {
```

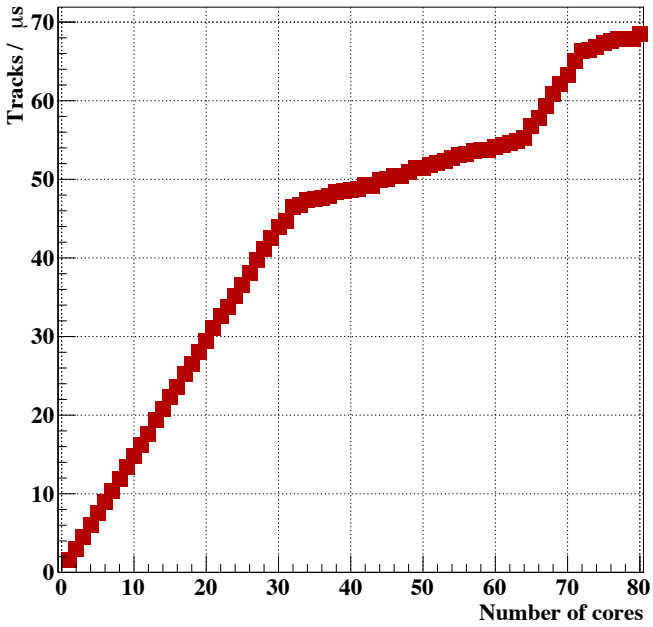
Part of the source code of Solution/Fit.cxx

```
706   int i = get_global_id(0);
707   TrackV curTrack = t[i];
708   Station vStations_local[10];
709   int iStation = 0;
710   for(iStation = 0; iStation<*NStations; iStation++ )
711   {
712     vStations_local[iStation] = vStations[iStation];
713     // printf("cl: %f %f \n", vStations_local[iStation].z.x,
      vStations[iStation].z.x);
714     // printf("cl: %f %f \n", vStations_local[iStation].thick.x,
      vStations[iStation].thick.x);
715     // printf("cl: %f %f \n", vStations_local[iStation].zhit.x,
      vStations[iStation].zhit.x);
716     // printf("cl: %f %f \n", vStations_local[iStation].RL.x,
      vStations[iStation].RL.x);
717     // printf("cl: %f %f \n", vStations_local[iStation].RadThick.x,
      vStations[iStation].RadThick.x);
718     // printf("cl: %f %f \n", vStations_local[iStation].logRadThick.x,
      vStations[iStation].logRadThick.x);
719     // printf("cl: %f %f \n", vStations_local[iStation].Sigma2.x,
      vStations[iStation].Sigma2.x);
720     // printf("cl: %f %f \n", vStations_local[iStation].Sigma.x,
      vStations[iStation].Sigma.x);
721     // printf("cl: %f %f \n", vStations_local[iStation].Sy.x,
      vStations[iStation].Sy.x);
722   }
723   int iTimes = 0;
724
725   for(iTimes = 0; iTimes<*NFits; iTimes++)
726      Fit(curTrack,vStations_local,*NStations);
727 //  if(i==0) printf("cl: %f %f %f %f ", curTrack.T[0].x, curTrack.T[5].x,
      curTrack.C.C00.x, curTrack.C.C44.x);
728
729   t[i] = curTrack;
730 }
```



When plotting the fitting results the picture should almost as the etalon one, which is obtained with the **single** binary, the difference is caused by different optimisation of the compilers:

The scalability with OpenCL looks like follows:



The platos with smaller slope are caused by hyper threading.

4.1. OpenCL on GPU

The exercises are at /home/kulakov/Exersises/7_OpenCL
One needs to create a folder at the Students folder and to copy the exercises into it. This will be your working folder.

To setup the environment run: **. ~/setOCL.sh**
To use local GPU in your programs you need to set: **export DISPLAY=:0**. It tells to the X11 applications, which screen it should run on. The format of the DISPLAY variable is hostname:display. The omitted hostname means that the local display will be used.

7_OpenCL/3_First_GPU: description

One needs to run on GPU the simdized example from the previous exercise (main3.cpp).

7_OpenCL/3_First_GPU: solution

To use GPU the following two changes must be done:

1. Change device type from **CL_DEVICE_TYPE_CPU** to **CL_DEVICE_TYPE_GPU**.
2. **Local_item_size** must be set correspondingly to the system. The maximum number of tasks, that the compute unit can proceed, is an appropriate value: **size_t local_item_size = 256**.

Scalar:

	Typical output after the solution
	Parallel time = 0.519 ms Scalar time = 0.006 ms

SIMD:

	Typical output after the solution
	Parallel time = 0.532 ms Scalar time = 0.009 ms

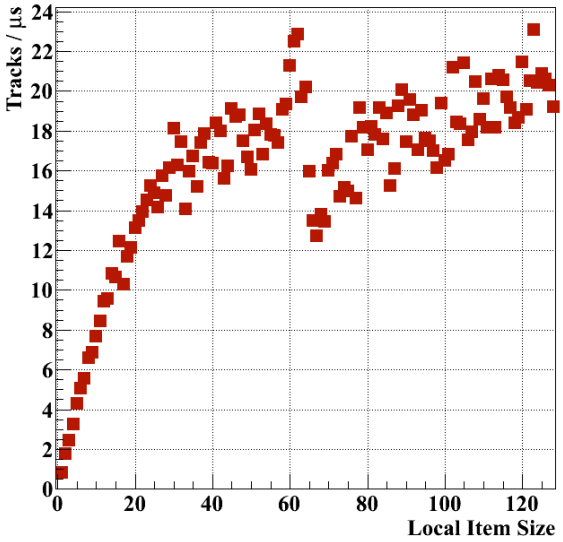
We see the negative speed up, because the tasks are very small.

7_OpenCL/4_SIMDKF_opencl1.2_vector_GPU

This exercise is the same as the solution of the exercise in the previous week.
Scalability can not be measured as before on GPU, since it can not be divided on sub-devices. The most important for GPU is a speed dependence on local item size, since load of the system depends on it. It is proposed to measure the dependence up to **local_item_size**=128, since the local memory resources are limited.
Since only one work group can be run only on one computing unit, we must see a scalable dependence up to the maximum number of threads on a computing unit (16 SIMD threads) - it will show gradual load of the computing unit. Ones the local item size exceeds the computing unit capacity, the whole working group will be divided in larger sub-tasks, this will lead to overhead decrease. Therefore, a stair-like dependence with gradually achieved saturation must be observed.

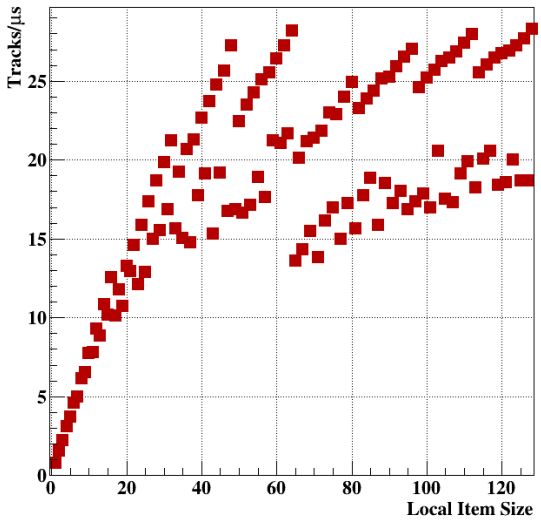
To measure the scalability:

- set display to local: **export DISPLAY=:0**
- run the program with 1 thread: **./opencl 1**
- go to the folder **TimeHisto** and measure the scalability:
- run bash script to collect the data: **. make_data_opencl_GPU.sh**
- if you get warnings in output file (**make_opencl_GPU_time.dat**) from GPU, replace them with void string. You can use **ctrl-** command in **nano** to perform the replacement;
- set display to standard: **export DISPLAY=localhost:11.0**
- plot the scalability: **root -l make_timehisto_stat_complex_opencl_GPU.C** .



The observed scalability shows linear increase in speed up to 16, then saturation is achieved as predicted.

7_OpenCL/5_SIMDKF_opencl1.1_vector_GPU



This exercise is changed to OpenCL1.1 to make it more portable (for instance, one can not run opencl1.2 on NVIDIA GPUs). OpenCL1.1 is C based, therefore all object oriented constructs were excluded from the code.
The measuring procedure for scalability is the same.
The observed scalability shows linear increase in speed up to 16, then each next 16-items-region (17-32, 33-48, …) shows the same dependence. Some of the measurements in the picture a lower, possible reason is particle load of the system by other users. Here we can see a predicted stair like dependence: each time, when the local item size is not dividable by the computing unit size, we use it inefficiently, the tasks will have different size and at the final stages some parts of the computing unit will be idle.

HPC Practical Course
Part 4.2

Intel Xeon Phi

V. Akishina, I. Kisel,
I. Kulakov, M. Zyzak

Goethe University of Frankfurt am Main

02 Jul 2014

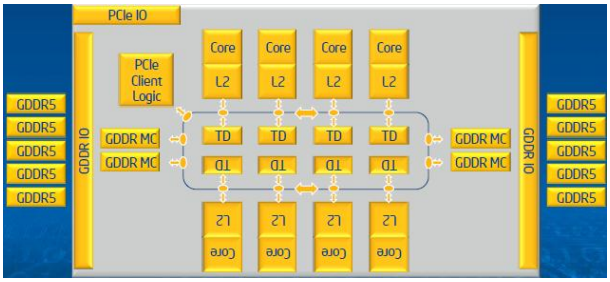
xeon-phi-dev Server

Intel Xeon CPU E5-2680

Intel Xeon Phi

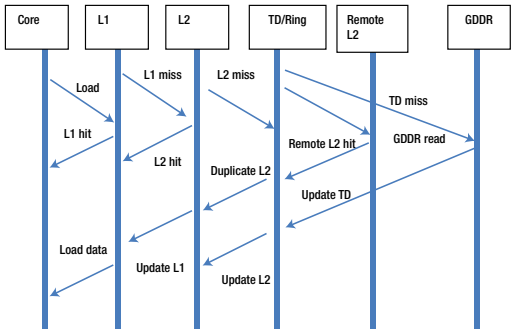
Intel Xeon Phi

Intel Xeon Phi



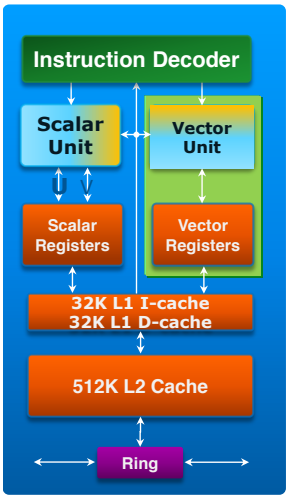
- 60 physical cores
- 4 hardware threads per core
- 512-bit vector registers (16 floats)
- Cores are connected through the bidirectional ring bus
- 8 GB of the GDDR5 memory:
  - 16 32-bit channels
  - up to 352 GB/s maximum memory bandwidth
  - 8 distributed memory controllers
- L1 cache latency is about 3 cycles, L2 - down to 14-15 cycles

Core/VPU Load Operation



- The core or VPU generates load signal.
- If data is not in L1, local L2 is checked.
- On a local L2 miss, a lookup to the TD (tag directory) happens.
- TD looks over remote L2. If the data are found in the L2 cache, the data are returned to the requesting core through an L1 update.
- If not found in the L2 cache, the data have to be fetched from the GDDR memory to the L2 cache through the memory controller.

Core Structure



- **Scalar Unit:**
  - Two pipelines
  - Scalar pipeline 1 clock latency
- **Vector Unit:**
  - 32 512-bit vector registers per thread
  - 8 16-bit mask registers per thread
  - Most operations: 4-cycle latency, 1-cycle throughput
  - Does not support MMX, SSE or AVX instruction set, supports only IMIC instructions

Materials: presentations of Klaus-Dieter Oertel, Software and Services Group, Intel Corporation

Exercise 1. Pi

- Compile the program with SSE instructions and run the program on CPU:
  - icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -msse -openmp -DHOST -o piSSE -lVc
  - ./piSSE 32
- Check the program on XeonPhi:
  - icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -mmic -openmp -L/home/mzyzak/Vc/vc/build/lib -o piMIC -lVc_MIC
  - . runMIC.sh 240
- Optimize the program for the maximum parallelisation speedup.
- Vectorize the CalculatePiSIMD() function.
- Check the speedup of vectorization and total speedup for SSE instructions on CPU.
- Compile the program with AVX instructions and run the program on CPU:
  - icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -mavx -openmp -DHOST -o piAVX -lVc
  - ./piAVX 32
- Check the speedup of vectorization and total speedup on Xeon Phi.
- Check the speedup from hyper threading for the program on a one core of Xeon Phi.

Exercise 2. SIMD KF on Xeon Phi

- Plot scalability vs number of cores on a one Xeon Phi card:
  - . make_data_omp_mic0.sh or . make_data_omp_mic1.sh
  - root -l -b -q plotMic0Scalability.C or root -l -b -q plotMic1Scalability.C
- Check speedup from hyper threading
- Plot total scalability on CPU with two Xeon Phi cards:
  - . make_data_omp_local_mic0_mic1.sh
  - root -l -b -q plotCombinedScalability.C
- Check the SIMD scalability (vs SIMD width) on CPU:
  - . make_data_singleVc_host.sh
  - root -l -b -q make_timehisto_SSE.C
  - . make_data_avxVc_host.sh
  - root -l -b -q make_timehisto_AVX.C
- Check the SIMD scalability (vs SIMD width) on Xeon Phi:
  - . make_data_singleVc_mic.sh
  - root -l -b -q make_timehisto_IMIC.C
- Explain the last plot. Make the scalability vs SIMD width linear.

4.2. Intel Xeon Phi

The exercises are at /micfs/ikulakov/Exersices/8_XeonPhi
One needs to create a folder in the Students folder and to copy the exercises into it. This will be your working folder.

To setup the environment run: **. /home/mzyzak/SetHost.sh**

8_XeonPhi/1_PI: description

In this exercise one needs to maximally parallelize the Pi exercise (using Vc, openMP and pthread) and test it on the CPU with AVX instructions (8 entries per vector) and on the Phi system. The ICC compiler has to be used on this server.

Part of the source code of pi.cpp

```
53 #pragma omp parallel for reduction(+ : sum) private(i,x) num_threads(nThreads)
54   for (i=1;i<= num_steps; i++){
55     x = (i-0.5)*step;
56     sum = sum + 4.0/(1.0+x*x);
57   }
58
59   pi = step * sum;
```

To compile the example with SSE instructions and run it use:
icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -msse -openmp -DHOST -o piSSE -lVc; ./piSSE
To compile the example with SSE instructions and run it use:
icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -mavx -openmp -DHOST -o piAVX -lVc; ./piAVX

To compile the example for Phi and run it use:
icc pi.cpp -inline-forceinline -std=c++11 -O3 -g -mmic -openmp -L/home/mzyzak/Vc/vc/build/lib -o piMIC -lVc_MIC; . runMIC.sh 240
The script **runMIC.sh** logins to mic0 card, copies executable there and runs it.

Part of the source code of runMIC.sh

```
 3 echo "Start"
 4
 5 PREF0=`pwd`
 6
 7 echo "Initializing data on the MIC"
 8 ssh mic0 "
 9   export LD_LIBRARY_PATH="/micfs/mzyzak/"
10
11   cp $PREF0/piMIC .
12   ./piMIC ${1}
13   rm -rf a.out
14   exit
15 "
16
17 echo "Done"
```

8_XeonPhi/1_PI: solution

Since each iteration has similar calculations, it is easy to vectorize the procedure. Loop iterations are grouped by vecN iterations in a group, then vecN results are summed together.

Part of the source code of pi_solution.cpp

```
68   float_v x, sum[nThreads];
69   double start_time;
70
71 #ifdef HOST
72   long num_steps = 10000000 * nThreads;
73 #else
74   long num_steps = 10000000 * nThreads;
75 #endif
76
77   float_v step = 1.0f/(float) num_steps;
78
79   start_time = omp_get_wtime();
80
81   for(int iTh=0; iTh<nThreads; iTh++)
82   sum[iTh]=0.f;
83
84   float_v index(int_v::IndexesFromZero());
85   index -= 0.5f;
86
87 #pragma omp parallel for private(i,x) num_threads(nThreads)
88   for (i=1;i<= num_steps; i+= vecN){
89     x = ( float_v(i) + index )*step;
90     sum[omp_get_thread_num()] += float_v(4.0f)/(float_v(1.0f)+x*x);
91   }
92
93   float_v sumSIMD=0.f;
94   for(int iTh=0; iTh<nThreads; iTh++)
95   sumSIMD += sum[iTh];
96
97   double sumScalar = 0.;
98   for(int iV=0; iV<vecN; iV++)
99   sumScalar += sumSIMD[iV];
100
101  pi = step[0] * sumScalar;
```

The program is already parallelized between cores using openMP, but the cores load can be increased using set_affinity, therefore one needs to add the code, which will fix each thread on its specific core:

Part of the source code of pi_solution.cpp

```
129 #pragma omp parallel num_threads(nThreads)
130   {
131     int s;
132     cpu_set_t cpuset;
```

Part of the source code of pi_solution.cpp

```
133     int cpuId = threadNumberToCpuMap[omp_get_thread_num()];
134     pthread_t thread  = pthread_self();
135     CPU_ZERO( &cpuset );
136     CPU_SET( cpuId, &cpuset );
137     s = pthread_setaffinity_np( thread, sizeof( cpu_set_t ), &cpuset );
138     if ( s != 0 ) {
139       std::cout << " pthread_setaffinity_np " << std::endl;
140       handle_error_en( s, "pthread_setaffinity_np" );
141     }
142   }
```

Results obtained with SSE:

Typical output after the solution

```
Scalar:   pi is 3.13613 in 0.0524721 seconds
SIMD:     pi is 3.13614 in 0.01302 seconds, speedup is 4.0301
Parallel: pi is 3.13614 in 0.0130408 seconds, speedup is 4.02369
```

Expected speed up factor of 4 is achieved.
Results obtained with AVX:

Typical output after the solution

```
Scalar:   pi is 3.13956 in 0.0579562 seconds
SIMD:     pi is 3.13956 in 0.0133219 seconds, speedup is 4.35045
Parallel: pi is 3.13956 in 0.01331 seconds, speedup is 4.35435
```

AVX vectors have 8 entires, but speed up factor of 4.5 is achieved only. Some AVX operations are emulated with SSE, some are implemented in hardware, but take more cycles than corresponding SSE operations.
Results obtained with Phi:

Typical output after the solution

```
Initializing data on the MIC
Scalar:   pi is 3.14132 in 0.191382 seconds
SIMD:     pi is 3.14132 in 0.0480771 seconds, speedup is 3.98073
Parallel: pi is 3.14284 in 0.0002443 seconds, speedup is 783.39
```

Expected SIMD speed up factor is 16, a probable reason why we get 4 is that the scalar version is auto-vectorized.
To forbid auto-vectorization, one can recompile with **-no-vec** flag.

Typical output after the solution

```
Scalar:   pi is 3.09941 in 1.2027 seconds
SIMD:     pi is 3.14132 in 0.037642 seconds, speedup is 31.951
Parallel: pi is 3.14284 in 0.000241962 seconds, speedup is 4970.61
```

The rise of speed up to 32 and decrease of precision is possibly compiler related.

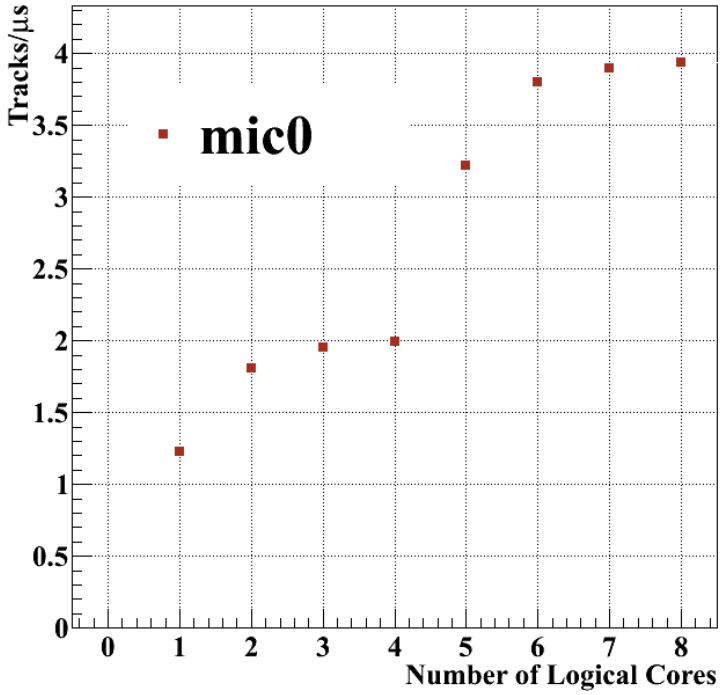
8_XeonPhi/2_SIMDKF: description

This exercise is the same as the solution of the exercise in the previous week.

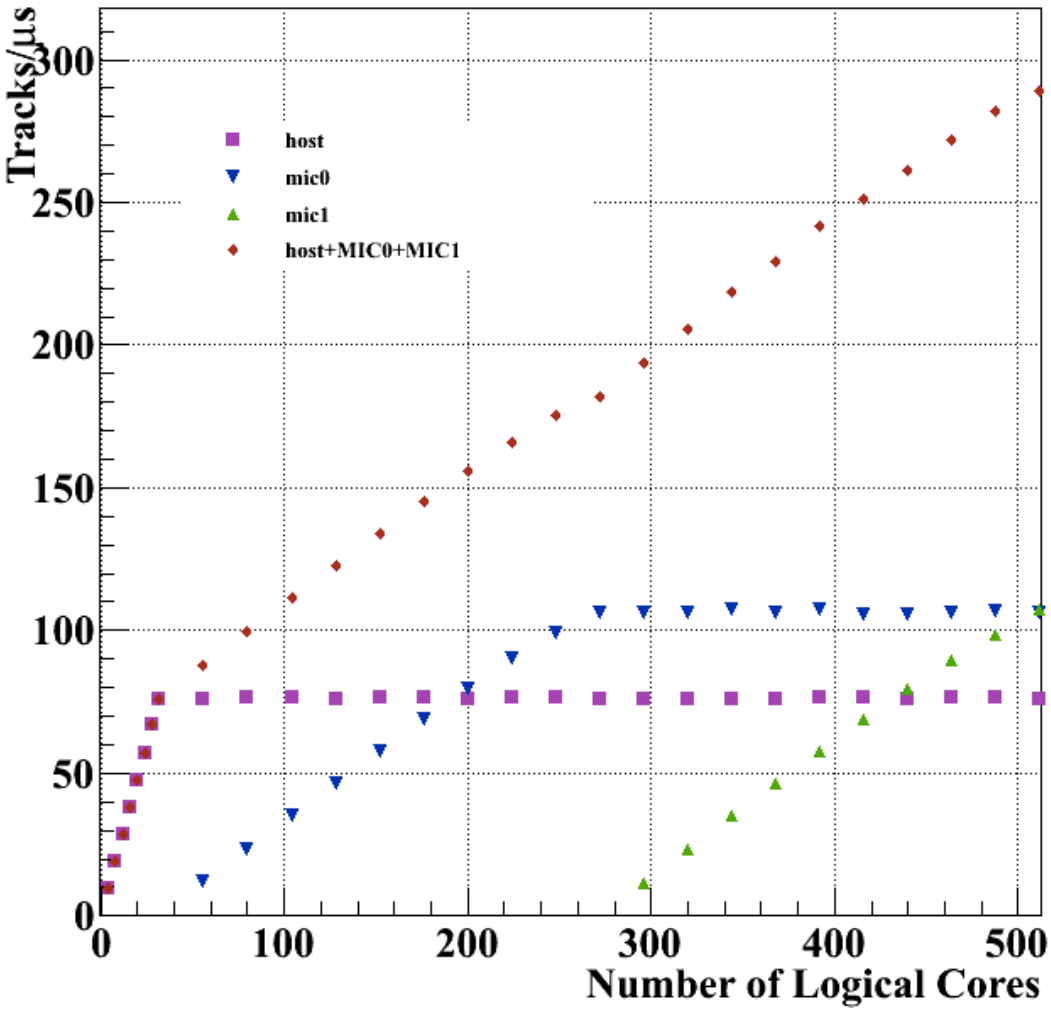
1. You need to measure the scalability on first 8 cores of the Phi card (takes ~1 minute):
- go to the folder **TimeHisto**
- run the bash script to collect the data: **. make_data_omp_mic0.sh**
- build the scalability: **root -l -q -b plotMic0Scalability.C**
- plot the picture: **evince ScalabilityMic0.pdf**

If the Phi card mic0 is busy, you can use another card mic1 by change 0 to 1 in the instructions above.

On the picture below you can see efficiency of the hyper-threaded cores: the 2-nd thread on the same physical core adds about 40% of speed, the 3-rd one - about 10%, the 4-rd - less than 5%.



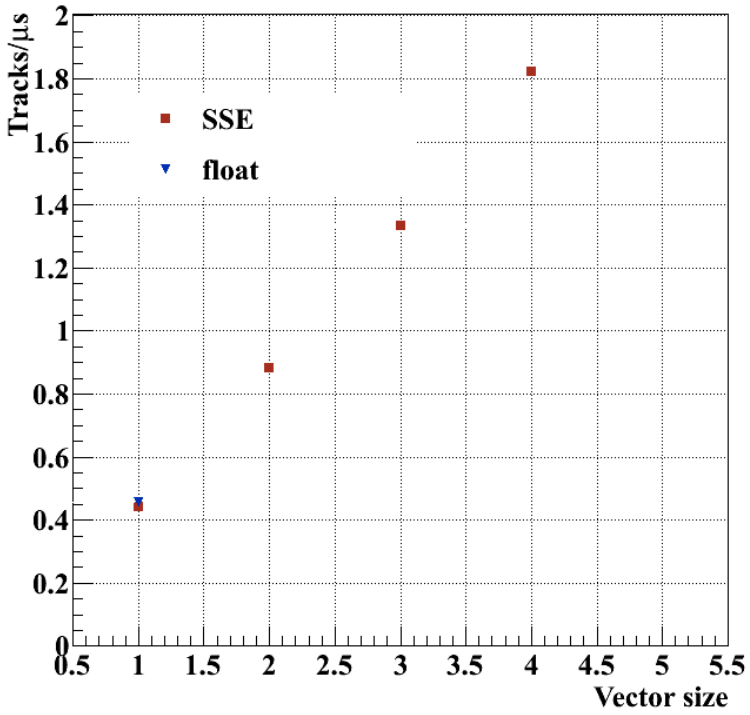
2. You need to measure the scalability on the whole system (CPU+2 Phi cards, takes ~4 minutes):
- go to the folder **TimeHisto**
- run the bash script to collect the data: **. make_data_omp_local_mic0_mic1.sh**
- build the scalability: **root -l -q -b plotCombinedScalability.C**
- plot the picture: **evince ScalabilityHostMic0Mic1.pdf**



The observed scalability shows linear increase in speed up to 32, which is the number of CPU cores, then linear increase up to 512 cores.

3. You need to measure the SSE SIMD-scalability on the CPU (takes ~0.5 minutes). SIMD-scalability is defined as dependence of speed on number of entries in SIMD-vector, which are used. To measure it you need to:

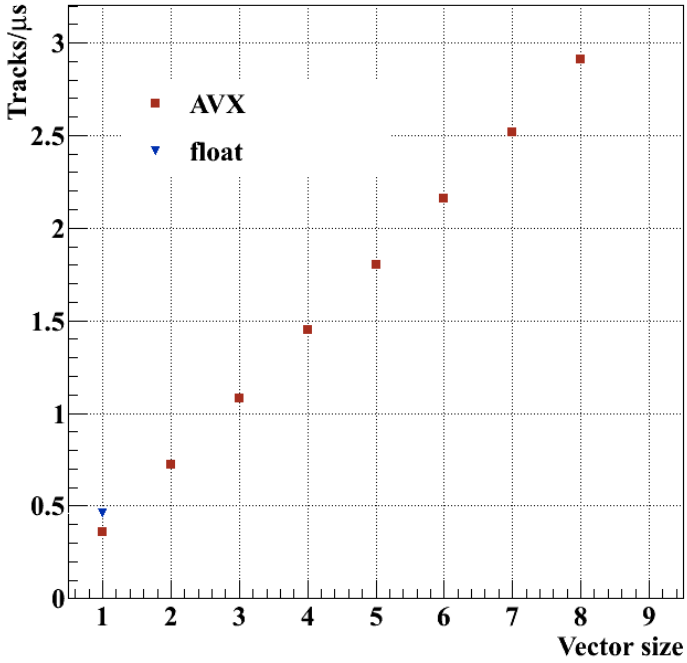
- go to the folder **TimeHisto**
- run the bash script to collect the data: **. make_data_singleVc_host.sh**
- build the scalability: **root -l -q -b make_timehisto_SSE.C**
- plot the picture: **evince Scalability_SSE.pdf**



The resulted dependence is linear as expected.

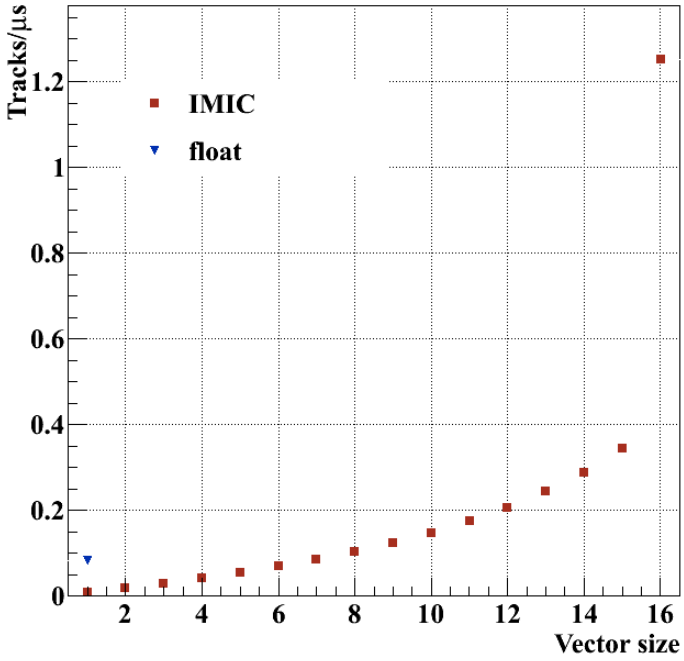
4. You need to measure the AVX SIMD-scalability on the CPU (takes ~0.5 minutes):

- go to the folder **TimeHisto**
- run the bash script to collect the data: **. make_data_avxVc_host.sh**
- build the scalability: **root -l -q -b make_timehisto_AVX.C**
- plot the picture: **evince Scalability_AVX.pdf**



The resulted dependence is linear as expected, but the speed up is slightly smaller than 8.

5. You need to measure the SIMD-scalability on Phi (takes ~5 minutes):
- go to the folder **TimeHisto**
- run the bash script to collect the data: **. make_data_singleVc_mic.sh**
- build the scalability: **root -l -q -b make_timehisto_IMIC.C**
- plot the picture: **evince Scalability_IMIC.pdf**



The resulted dependence is not linear. This is a task for you to fix the problem.

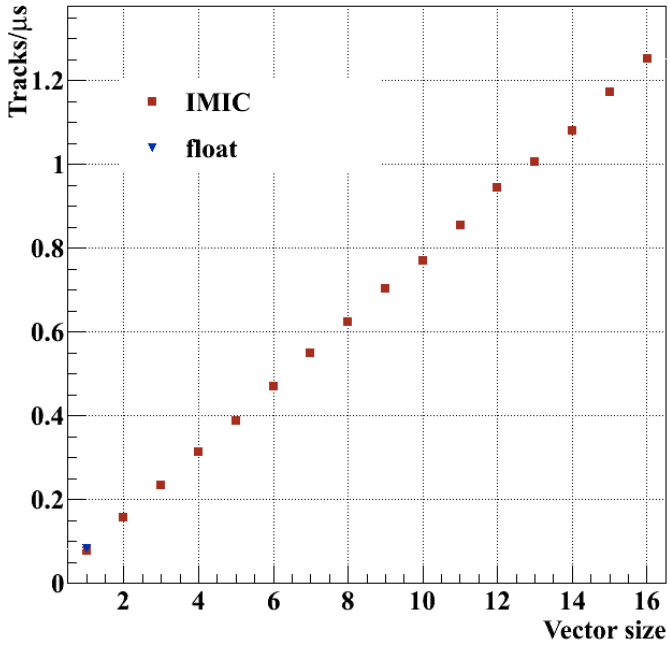
8_XeonPhi/2_SIMDKF: solution

The reason of the nonlinear SIMD scalability is that the unused entries in vectors are not initialized. When operating with non-initialized values floating point exceptions can appear. For example, division by zero or operations with NaN. Such exceptions need additional time to handle.

To fix this problem one can initialize all unused entries by a value of the first used entry.

Part of the source code of Fix_solution.cxx

```
365     for(int it=vecN; it<Fvec_t::Size; it++ ){
366       for( int ista=0; ista<NStations; ista++ ){
367         hxmem[ista][it] = hxmem[ista][0];
368         hymem[ista][it] = hymem[ista][0];
369         hwmem[ista][it] = 1.;
370       }
371     }
```



The resulting scalability is linear, speed up is close to 16.

References

1. Unix shell

   https://pangea.stanford.edu/computing/unix/shell/commands.php
   http://www.gnu.org/software/bash/manual/bashref.html

2. C++

   http://en.cppreference.com/w/Main_Page
   Bjarne Stroustrup “Programming: Principles and Practice Using C++”
   Book by Bjarne Stroustrup “The C++ Programming Language”.

3. SIMD

   http://msdn.microsoft.com/de-de/library/y0dh78ez(v=vs.90).aspx
   Intel® 64 and IA-32 Architectures Optimization Reference Manual

4. Kalman filter track fit

   http://web-docs.gsi.de/~ikisel/17_CPC_178_2008.pdf

5. Vc

   http://code.compeng.uni-frankfurt.de/projects/vc

6. OpenMP

   http://openmp.org/wp/
   http://www.openmp.org/mp-documents/spec30.pdf

7. ITBB

   http://threadingbuildingblocks.org
   O'Reilly Media, “Intel Threading Building Blocks”

8. OpenCL

   http://www.khronos.org/opencl/
   http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
   http://de.geforce.com/hardware/desktop-gpus/geforce-gtx-480/architecture

9. Xeon Phi

   http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
   https://software.intel.com/sites/default/files/managed/f5/60/intel-xeon-phi-coprocessor-quick-start-developers-guide-mpss-3.2.pdf