MTH789P - Trading and Risk Systems Development

Lecture XVIII. High Performance Computing

Dr. Pedro Vergel Eleuterio

School of Mathematical Sciences
Queen Mary University of London

Outline I

1. High Performance Computing
   - What is it?
   - Examples
2. Moore's law
3. Parallelism
   - Fine/Coarse grain parallelization
   - Load balancing
   - Embarrasingly Parallel Problems
   - Compute to Global Memory Access Ratio
   - Flynn's taxonomy
   - Amdahl's law
   - FLOPS
4. Pipelining
5. SIMD Registers
6. Multithreading

Outline II

- Hyperthreading
- An example with C++11 threads
- An example with OpenMP

7. Grid computing

8. Graphics Processing Units
    - Programming GPUs
    - Example: coding in OpenCL

9. Field Programmable Gate Arrays
    - Programming FPGAs

10. Specialised software
    - KdB

High Performance Computing

What is it?

High Performance Computing (HPC) refers to a set of techniques and strategies to execute CPU intensive processes that would otherwise take too long to execute.
Examples uses of HPC in finance:

- FX and Equity trading takes place much faster than Interest Rate and Credit derivatives so technology requirements and priorities can be very different. In FX the ability to run PV and Greeks for a portfolio of thousands of trades in less than 5 minutes can provide a trading operation with a real competitive edge.
- CVA (Credit Value Adjustment) have become key to financial institutions due to new regulatory requirements. This can require to run millions of derivatives valuations that can become impossible in traditional technological frameworks.

High Performance Computing

Examples

Some of the techniques that are used are:

- Grid computing
- GPUs
- FPGAs
- Multithreading

Moore’s law

- In 1965 George Moore speculated that the density of transistors in chips doubled every two years.
- This conjecture has been verified for the last 40 years. As a result memory capacity and execution speed have also doubled every two years.
- However with the current 22nm architectures this progression is being questioned as parts of the circuits are now just around 0.5nm, two or three atoms wide (a Silicon atom is 0.234nm wide).
- There are some discussions of 16nm, 14nm, 11nm and 10nm chips. E.g. NVIDIA chief scientist claims they will be able to use 11nm chips in 2015. Intel’s CTO claims they will be have 10nm chips in 2015.
- However the it is not clear that Moore’s law is sustainable.
- For this reason technology has moved in the direction of parallelization: multicores, SIMD registers, GPUs . . .

Parallelism

- Most of HPC techniques are based on the idea of parallelization: performing tasks simultaneously on different processing units that would otherwise be executed sequentially.
- For example we could run different Monte Carlo simulations on different computers, use several cores inside a computer, etc.
- Writing parallel programs is not easy as we have to think on how to optimally break the code, transfer input and output, add the different results together etc.

Parallelism

Fine/Coarse grain parallelization

Depending on what is the size of the tasks that is parallelized we talk about coarse or fine parallelization.

- In *coarse grain parallelization* we break the task in large chunks. For example, we break the valuation of a portfolio in valuation of single trades, distribute them across different computers and then collect the results.
- In *fine grained parallelism* we break the task in very small tasks, for example: we group every 4 multiplications of `doubles`'s in a loop in one by using the new 256 bit AVX registers (`YMM0`,. . . , `YMM7`) in the Intel Sandy Bridge chips (i3/i5/i7 Core) that can multiply simultaneously 4 doubles.

The ideal level of granularity depends on the computation to communication ratio. If the cost of communication is expensive then it is better to send off large amounts of calculation.

Parallelism

Load balancing

- An important factor in parallelization is *load balancing*.
- This means ensuring that all processing units are receiving similar amounts of work.
- For example: if you have 100 computers to value 100 vanilla trades and 100 complex exotic (slow) trades. You would not want to send all exotic trade to 50 of the CPUs and all vanilla trades to the other 50 CPUs, as the latter set would be finished and idle whilst the first 50 CPUs are still working on the more complex deals. Load balancing is the art of optimally distributing tasks according to their computational load.

Parallelism

Embarrasingly Parallel Problems

Often parallelization require us to completely rethink the numerical algorithms we use for valuations.

In some occasions parallelization is straightforward as the task at hand already consists of separate tasks, we refer to these as *embarrasingly parallelisable* problems.

http://en.wikipedia.org/wiki/Embarrassingly_parallel

Parallelism

Compute to Global Memory Access Ratio

- Parallelizing a task often incurs an overhead. For example we might need to send some tasks to the cores in a compute grid through an Ethernet connection.
- If the cost of setting up the technology and transferring the data is larger than the benefits of parallelizing the calculation then the parallelization strategy is useless.
- For this reason we need to monitor the Compute to Global Memory Access Ratio (CGMA Ratio).

Parallelism

Flynn's taxonomy

Useful terminology to categorize several approaches to parallelism is Flynn's taxonomy:

	Same data	Multiple data
Same instruction	*SISD*	SIMD
Multiple instructions	*MISD*	MIMD

Of these SISD is the same as plain normal execution and MISD is *almost* nonsense (used for critical processes that are replicated for safety, see `http://en.wikipedia.org/wiki/MISD`).
So effectively SIMD and MIMD are the only terms used.
SIMD is described as vector processing and MIMD as concurrent processing.
See `http://en.wikipedia.org/wiki/Flynn%27s_taxonomy` for more information.

Parallelism

Amdahl's law

This law states that if $x\%$ of our program execution time can be parallelized then by using N processors to parallelize that code our code will take

$$\frac{1}{N}x\% + (1 - x\%)$$

times the time it takes with no parallelization. So the speed-up factor will be:

$$\frac{1}{\frac{1}{N}x\% + (1 - x\%)}.$$

For example if 90% of the code is parallelizable then the speed-up factor is

$$\frac{1}{\frac{1}{N}0.9 + 0.1},$$

which as $N \to \infty$ grows to 10. So at most we can make our program run 10 times faster.

Parallelism

FLOPS

A measure of performance of a computer is the number of *FLOPS* it can achieve.

- FLOPS stands for *"FLoating Point oPerations Per Second"*. A floating point operation refers to an addition, subtraction, multiplication or division of floating point numbers.
- Most modern computers can achieve 4 floating point operations per cycle by using pipelining (see next slide). For a computer running at 3.3Gz this means it can achieve $3.3\text{m}\times 4$ floating point operations per second per core. For a dual core CPU this is roughly 26 Gigaflops (GFLOPS).
- GFLOPS stands for 10^6 FLOPS.

Parallelism

FLOPS

- Likewise we have:

exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

- Generally speaking:
    - Modern day PCs can sustain tens of GFLOPS.
    - GPUs can sustain TFLOPS,
    - Large scale parallel computers run in the order of PFLOPS. The fastest computer is currently (as of June 2013) China's Tianhe-2 running at 33.86 PetaFLOPS.
- Several participants in the HPC arena have set reaching ExaFLOPS computing capability by 2018 as the next big challenge for the industry.

Pipelining

- The lowest level of software optimization is called *pipelining*.
- It consists of ordering assembly instructions in such a way that consecutive instructions involve different parts of the CPU and therefore when the CPU breaks them in micro-ops, they can be executed almost simultaneously.
- For example an addition can be performed simultaneously to a memory store operation (of data not involved in the addition). So a good compiler will reorder our code in such a way.
- Normal persons will not get involved in writing any assembly code or designing any pipelining. The compiler will take care of implementing these optimizations when converting our code to assembly.

SIMD Registers

The next degree in low level parallelism is the use the SIMD registers in the CPU.

- This technology started in 1997 as MMX (MultiMedia eXtensions). A set of 8 64 bit registers called `MM0` to `MM7` each of which could hold and process in parallel 8 8 bit integers, or 4 16 bit integers, or 2 32 bit integers, or 1 64 bit integer.
- In 1999 Intel released the SSE (Streaming SIMD Extensions) which implemented 8 registers `XMM0` to `XMM7` each of them holding 128 bits that could be used as a set of 2 doubles or 4 floats to be operated in parallel.

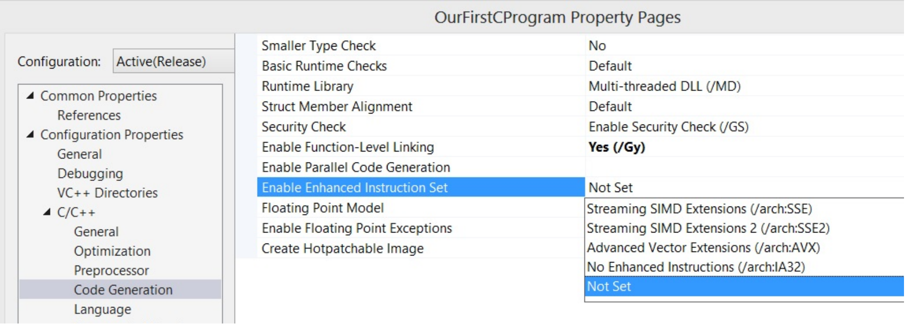
SIMD Registers

- In 2011 Intel released AVX (Advanced Vector eXtensions), a set of 8 (16 if running in 64 bit mode) 256 bit registers named `YMM0` to `YMM7` that can operate as a vector of 4 doubles or 8 floats that can be added, multiplied etc. in parallel.
- Intel is scheduled to release AVX-512 in 2015. This will consist of 32 vector registers `ZMM0` to `ZMM31` of size 512 bits that can hold 8 doubles and 16 floats an can be operated in parallel.

Writing quant calculations optimized by using AVX could make you big bucks.

SIMD Registers

Note that the low level of optimisation alluded to in the previous slide is often accomplished by the C/C++ compiler and not often by us writing assembly code.
To do so you need to understand the relevant switches in your C/C++ compiler. For example in Visual Studio you can request use of AVX or SSE instructions as follows:



OurFirstCProgram Property Pages

Configuration: Active(Release)

- Common Properties
  - References
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language

Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	Multi-threaded DLL (/MD)
Struct Member Alignment	Default
Security Check	Enable Security Check (/GS)
Enable Function-Level Linking	**Yes (/Gy)**
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	Not Set
Floating Point Model	
Enable Floating Point Exceptions	
Create Hotpatchable Image	

Streaming SIMD Extensions (/arch:SSE)
Streaming SIMD Extensions 2 (/arch:SSE2)
Advanced Vector Extensions (/arch:AVX)
No Enhanced Instructions (/arch:IA32)
Not Set

Writing optimizable C/C++

We have mentioned that very low levels of optimization (pipelining and use of SIMD registers) are most often best left to the compiler. However the way we write C/C++ might impact the ability of the compiler to optimize.

- Unrolling loops (i.e. writing them in a way that 2 or 4 iterations are performed inside the scope of the `for` loop will help the compiler compile this into SIMD instructions.
- Avoiding the use of `if` statements in a loop that can break predictive execution in the CPU.
- Using arrays of structs rather that structs of arrays or separate arrays so as to allow data that is processed together to be stored in memory together so that they can be optimally cached in the CPU.

The best reference on this and related matters is Agner Fog's web `www.agner.org/optimize/`.

Multithreading

- The next degree in low level parallelism is *multithreading*.
- This means breaking our code execution so that it utilizes several cores (CPUs) that our machine might have. For example on a quad core machine we can run 4 calculations concurrently in each of the cores.
- Each of these concurrent programs is called a *thread*.
- Standard C++ does not support threads. You need to use external libraries to support this. Several options are: POSIX, boost, . . .
- OpenMP and Intel's Threading Building Blocks are libraries that allows you to multithread your code for certain limited tasks in a simple manner without having to deal with threads directly.
- The new C++ standard, C++11, (released August 2011) specifies functionality to create threads by means of the class `std::thread`.

Multithreading

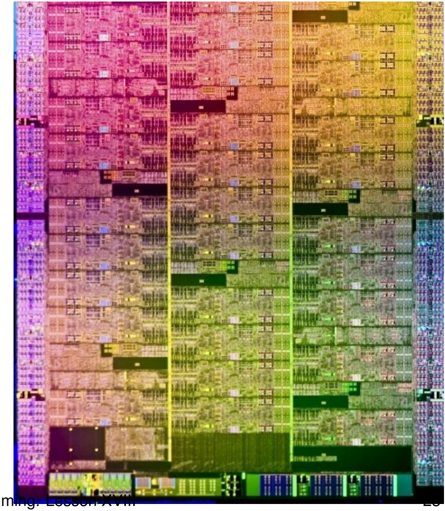
- Multithreading can be very difficult to debug as several threads will be executing simultaneously so it will be difficult to follow the execution of one single thread.
- Management of what data should a thread be able to write is very delicate. One can end up with *race conditions* whereby one thread can pollute data that is used by another thread, often in unpredictable manner which gives rise to difficult to reproduce and fix bugs.
- Code that can be made to work in threads is called *thread-safe*. Thread safe code is generally code that does not write on global or static variables.

Because of the difficulties in debugging code, development teams often are very cautions with regards to multithreading. If you become an expert in multithreading you can have a big impact in such a team.

Multithreading

The Xeon Phi processor

- In November 2012 Intel released the Xeon Phi co-processor. It holds up to 61 cores and can provide 1.2 TFLOPS of double precision performance.
- It comes in series 3100 (57 cores, 1.1GHz, 6Gb), 5100 (60 cores, 1.053GHz, 8Gb) and 7100 (61 cores, 1.238GHz, 16Gb).
- It is only supported by Linux.
- The record breaking 33.86 PFLOPS Tianhe-2 supercomputer has 16,000 nodes with three Xeon Phi and two Ivy Bridge chips per node totalling 3,120,000 cores.



Multithreading

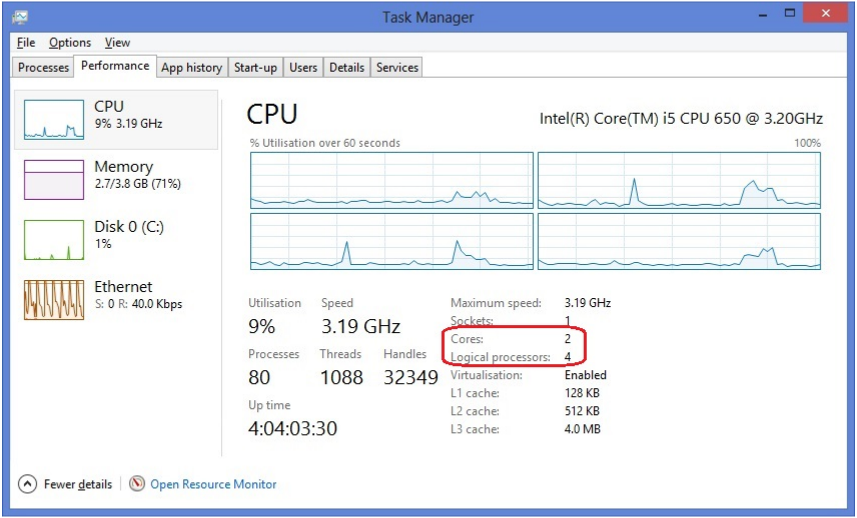
Hyperthreading

- *Hypethreading* is a hardware technique whereby the CPU can simulate having double the number of cores. It increases performance by around 30% (so not 2×!).
- You can switch it on/off from the BIOS menu at start-up.
- We use the expression *physical cores* vs. *logical cores* to distinguish between the real cores in the CPU and the ones simulated with hyperthreading.

Multithreading

Hyperthreading

Task manager will report the number of logical and physical cores:



CPU
9% 3.19 GHz

Memory
2.7/3.8 GB (71%)

Disk 0 (C:)
1%

Ethernet
S: 0 R: 40.0 Kbps

CPU

Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz

% Utilisation over 60 seconds 100%

Utilisation
9%

Speed
3.19 GHz

Processes
80

Threads
1088

Handles
32349

Up time
4:04:03:30

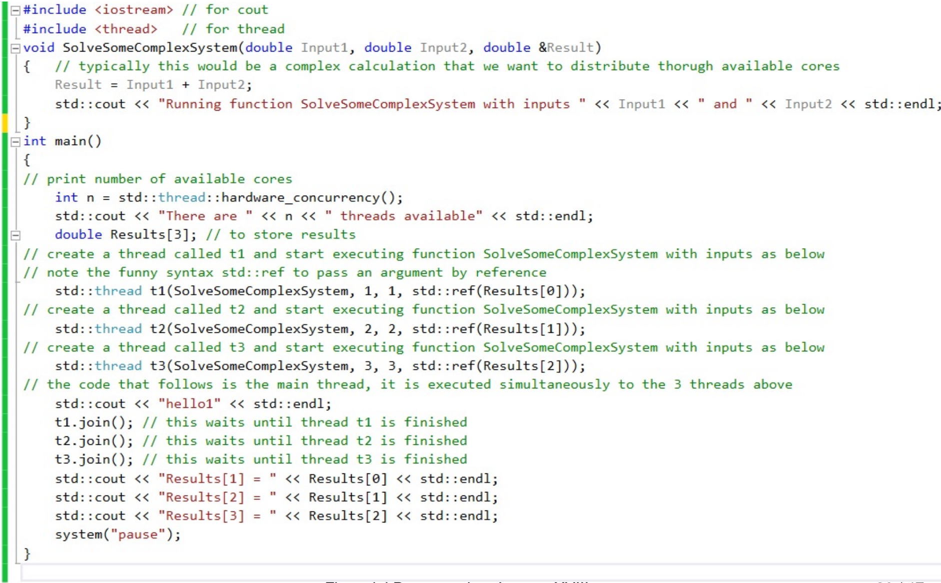
Maximum speed:	3.19 GHz
Sockets:	1
Cores:	2
Logical processors:	4
Virtualisation:	Enabled
L1 cache:	128 KB
L2 cache:	512 KB
L3 cache:	4.0 MB

Multithreading

An example with C++11 threads

Queen Mary University of London

School of Mathematical Sciences

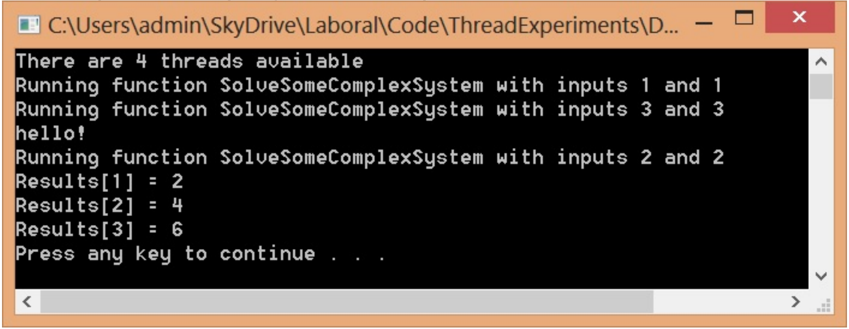


```
#include <iostream> // for cout
#include <thread>   // for thread
void SolveSomeComplexSystem(double Input1, double Input2, double &Result)
{   // typically this would be a complex calculation that we want to distribute thorugh available cores
    Result = Input1 + Input2;
    std::cout << "Running function SolveSomeComplexSystem with inputs " << Input1 << " and " << Input2 << std::endl;
}
int main()
{
// print number of available cores
    int n = std::thread::hardware_concurrency();
    std::cout << "There are " << n << " threads available" << std::endl;
    double Results[3]; // to store results
// create a thread called t1 and start executing function SolveSomeComplexSystem with inputs as below
// note the funny syntax std::ref to pass an argument by reference
    std::thread t1(SolveSomeComplexSystem, 1, 1, std::ref(Results[0]));
// create a thread called t2 and start executing function SolveSomeComplexSystem with inputs as below
    std::thread t2(SolveSomeComplexSystem, 2, 2, std::ref(Results[1]));
// create a thread called t3 and start executing function SolveSomeComplexSystem with inputs as below
    std::thread t3(SolveSomeComplexSystem, 3, 3, std::ref(Results[2]));
// the code that follows is the main thread, it is executed simultaneously to the 3 threads above
    std::cout << "hello1" << std::endl;
    t1.join(); // this waits until thread t1 is finished
    t2.join(); // this waits until thread t2 is finished
    t3.join(); // this waits until thread t3 is finished
    std::cout << "Results[1] = " << Results[0] << std::endl;
    std::cout << "Results[2] = " << Results[1] << std::endl;
    std::cout << "Results[3] = " << Results[2] << std::endl;
    system("pause");
}
```

Multithreading

An example with C++11 threads

- The output of the program in the previous slide is:



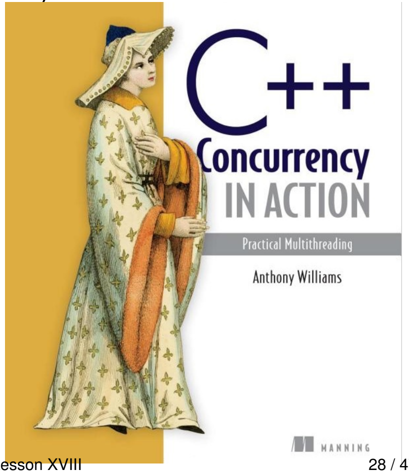
```
There are 4 threads available
Running function SolveSomeComplexSystem with inputs 1 and 1
Running function SolveSomeComplexSystem with inputs 3 and 3
hello!
Running function SolveSomeComplexSystem with inputs 2 and 2
Results[1] = 2
Results[2] = 4
Results[3] = 6
Press any key to continue . . .
```

- Note that the first lines are all mixed, this is because the 3 threads and the main thread are being executed simultaneously.
- The program in the previous slide uses the thread functionality offered by the C++11 standard which Visual Studio 2013 supports.

Multithreading

An example with C++11 threads

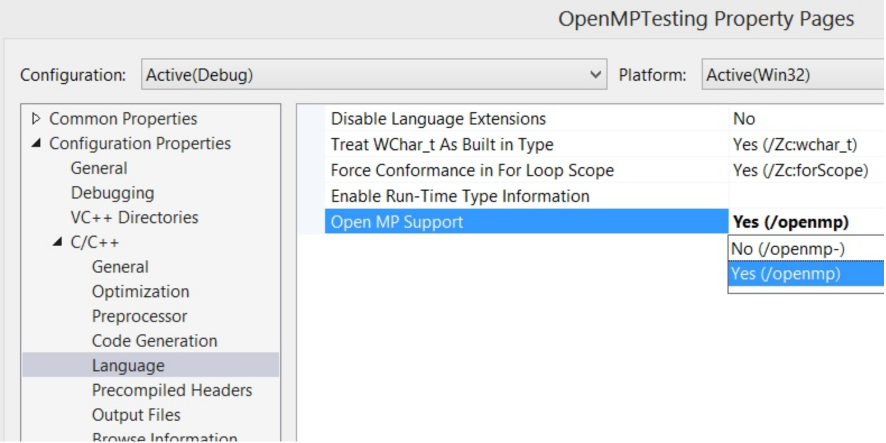
- Note that in the example above the instruction `std::thread::hardware_concurrency()` returned the number of threads (logical processors) rather than the number of physical cores.
- A book on C++11 threads is *C++ Concurrency in Action* by Anthony Williams
- **Homework:** Copy the code above into a project in Visual Studio and execute it. You might get slightly different results depending on the number of cores available. The order in which threads are executed is essentially random.



Multithreading

An example with OpenMP

To use OpenMP you first need to tell Visual Studio by PROJECT › Properties... › Configuration Properties › C/C++ › Language › OpenMP Support



OpenMPTesting Property Pages

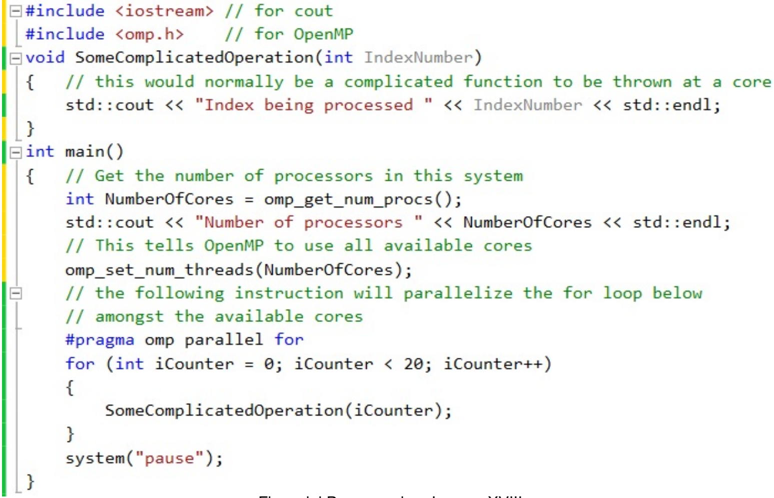
Configuration: Active(Debug)
Platform: Active(Win32)

Disable Language Extensions	No
Treat WChar_t As Built in Type	Yes (/Zc:wchar_t)
Force Conformance in For Loop Scope	Yes (/Zc:forScope)
Enable Run-Time Type Information	
Open MP Support	**Yes (/openmp)**

Multithreading

An example with OpenMP

The following code uses OpenMP to distribute a loop amongst different all available cores:

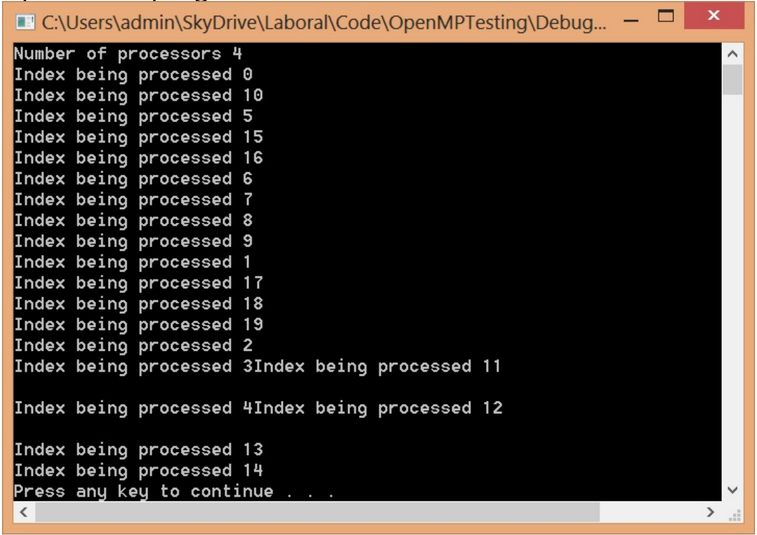


```
#include <iostream> // for cout
#include <omp.h>    // for OpenMP
void SomeComplicatedOperation(int IndexNumber)
{   // this would normally be a complicated function to be thrown at a core
    std::cout << "Index being processed " << IndexNumber << std::endl;
}
int main()
{   // Get the number of processors in this system
    int NumberOfCores = omp_get_num_procs();
    std::cout << "Number of processors " << NumberOfCores << std::endl;
    // This tells OpenMP to use all available cores
    omp_set_num_threads(NumberOfCores);
    // the following instruction will parallelize the for loop below
    // amongst the available cores
    #pragma omp parallel for
    for (int iCounter = 0; iCounter < 20; iCounter++)
    {
        SomeComplicatedOperation(iCounter);
    }
    system("pause");
}
```

Multithreading

An example with OpenMP

The output of the program above is as follows:



```
Number of processors 4
Index being processed 0
Index being processed 10
Index being processed 5
Index being processed 15
Index being processed 16
Index being processed 6
Index being processed 7
Index being processed 8
Index being processed 9
Index being processed 1
Index being processed 17
Index being processed 18
Index being processed 19
Index being processed 2
Index being processed 3Index being processed 11

Index being processed 4Index being processed 12

Index being processed 13
Index being processed 14
Press any key to continue . . .
```

Multithreading

An example with OpenMP

- Note the semi-random order of the instructions executed.
- OpenMP has detected 4 cores and has divided the 20 iterations in the loop amongst these cores.
- Note that 0,1,2,3,4 are in the right order as they will have been executed by core #1 in that order. Simultaneously core #2 has executed iterations 5,6,7,8,9, core #2 10,11,12,13,14 and core #4 15,16,17,18,19.
- **Homework:** Code this, compile and execute it, you might get different output depending on your computer configuration.

Grid computing

- Grid computing refers to schemes where the user computer (the client) sends a task to be distributed amongst a large amount of computers called nodes. It is sometimes referred to as *distributed computing*.
- In order to distribute tasks, send input data, collect and aggregate output we need software which is often called the grid middleware.
- Grids used in financial institutions can hold thousands of cores.
- Sometimes we might set up grids to utilize unused machines in offices at night or in disaster recovery sites. These are called *scavenged grids*. They can be problematic as the quality of the computers (size of RAM etc) might not be consistently high.

Grid computing

- An important aspect often overseen is to plan what to do if a calculation node fails. This could be due to a bug in the code or a hardware or configuration problem. It is important that from the outset an adequate infrastructure to deal with this is designed and put in place so that errors can be tracked, understood and debugged.

Graphics Processing Units

- GPU stands for Graphic Processing Unit. This are the processing units in graphics cards. High end GPUs can easily hold 300 cores to run parallel (SIMD) code.
- An important aspect of GPU programming is memory latency: the amount of time it takes to move data from the main memory to a GPU is not trivial. Therefore we need ensure the ratio (amount of processing)/(data transfer) is sufficiently high.
- The acronym GPGPU stands for General purpose GPU programming (i.e. as opposed to programming exclusively for graphics)

GPUs are programmed with languages specifically designed for them. The following are the standard languages:

1. CUDA is the language provided by the manufacturer NVIDIA. It is suitable to program NVIDIA cards which are used by a large part of the HPC sector.
2. OpenCL (Open Computer Language) is independent of any hardware provider and allows to distribute tasks to GPUs and multicore CPUs independently of the manufacturer.

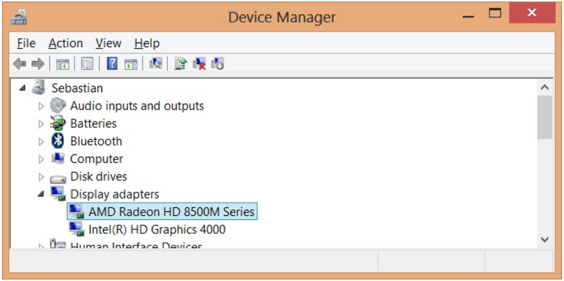
Both CUDA and OpenCL are similar to C/C++ with some differences specific to GPU programming.

As in multithreading the difficulties associated with debugging parallel code in addition to the requirement of delicate memory management can make efficient GPU programming challenging which can cause some financial institutions not to use this technology.

Graphics Processing Units

Example: coding in OpenCL

In order to establish whether your computer has a device that can support OpenCL, you need to open the Device Manager in the control panel



- Sebastian
  - Audio inputs and outputs
  - Batteries
  - Bluetooth
  - Computer
  - Disk drives
  - Display adapters
    - AMD Radeon HD 8500M Series
    - Intel(R) HD Graphics 4000
  - Human Interface Devices

Above we see an AMD Radeon graphics chip as well as a quad core CPU. Both of these support OpenCL.

Graphics Processing Units

Example: coding in OpenCL

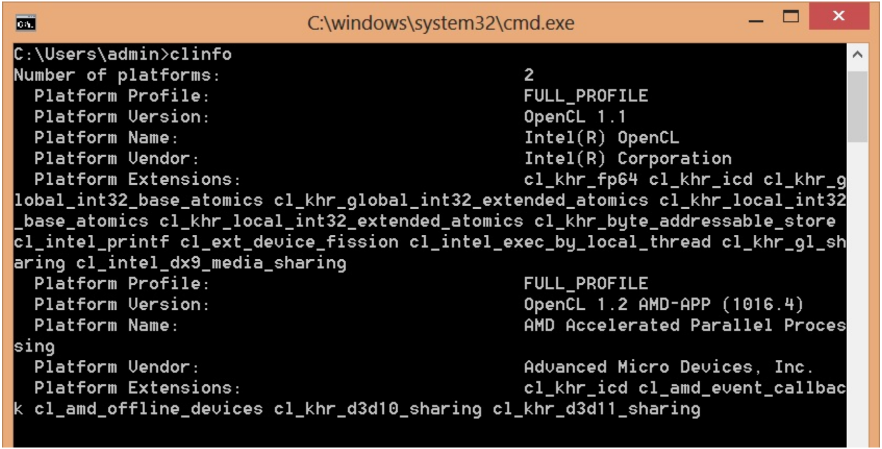
Visual Studio knows nothing about OpenCL. In order to program in OpenCL we need functionality that can allow Visual Studio to understand the OpenCL language.
To do this we download the OpenCL SDK (Software Development Kit) from:
http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/downloads/#one

Make sure you reboot your system after install, otherwise Visual Studio will not work with OpenCL.

Graphics Processing Units

Example: coding in OpenCL

We can now explore in more detail the available devices by opening a command window (⊞+R, then `cmd`+↵) and running the command `clinfo`



```
C:\Users\admin>clinfo
Number of platforms:                             2
  Platform Profile:                              FULL_PROFILE
  Platform Version:                              OpenCL 1.1
  Platform Name:                                 Intel(R) OpenCL
  Platform Vendor:                               Intel(R) Corporation
  Platform Extensions:                           cl_khr_fp64 cl_khr_icd cl_khr_g
lobal_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32
_base_atomics cl_khr_local_int32_extended_atomics cl_khr_byte_addressable_store
cl_intel_printf cl_ext_device_fission cl_intel_exec_by_local_thread cl_khr_gl_sh
aring cl_intel_dx9_media_sharing
  Platform Profile:                              FULL_PROFILE
  Platform Version:                              OpenCL 1.2 AMD-APP (1016.4)
  Platform Name:                                 AMD Accelerated Parallel Proces
sing
  Platform Vendor:                               Advanced Micro Devices, Inc.
  Platform Extensions:                           cl_khr_icd cl_amd_event_callbac
k cl_amd_offline_devices cl_khr_d3d10_sharing cl_khr_d3d11_sharing
```

Graphics Processing Units

Example: coding in OpenCL

`clinfo` provides a lot of information on our devices, for example whether it is a CPU or a GPU, amount of memory on the device, number of cores, support for double precision, etc

GGraphics Processing Units

Example: coding in OpenCL

The next task is to configure Visual Studio so that it can use OpenCL. To do this start a new empty project, then:

1. Go to PROJECT › Properties. . . › Configuration Properties › C/C++ › All Options › Additional Include Directories and add `$(AMDAPPSDKROOT)\include`. This tells (the compiler in) Visual Studio where to find header files that will advertise OpenCL functionality that will be provided elsewhere.
2. Go to PROJECT › Properties. . . › Configuration Properties › Linker › General › Additional Library Directories and add `$(AMDAPPSDKROOT)\lib\x86`.
3. Go to PROJECT › Properties. . . › Configuration Properties › Linker › Input › Additional Dependencies and add `OpenCL.lib`. These last two steps tell (the linker in) Visual Studio where to find the LIB files with the actual OpenCL functionality.

Graphics Processing Units

Example: coding in OpenCL

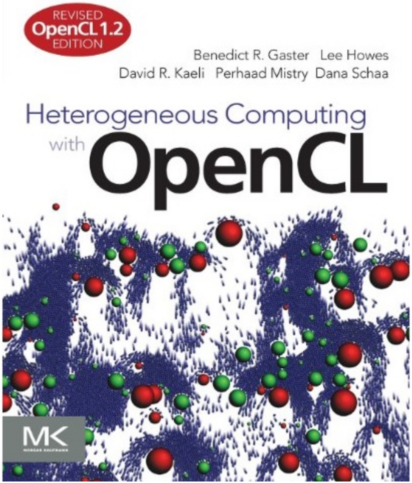
- You can now write OpenCL code in Visual Studio.
- As explained above OpenCL is more flexible than CUDA in that it works for several types of hardware (platforms) that can use OpenCL and there can be several GPU cards (devices), etc.
- OpenCL is therefore more verbose in that you need to inspect and choose amongst all these options (platforms, devices) and set up OpenCL programs (contexts) which contain OpenCL functions (kernels) etc.

For this reason we will not be able to display an OpenCL program on the slides (it would be too long).

Graphics Processing Units

Example: coding in OpenCL

See *Heterogeneous Computing with OpenCL* by B. R. Gaster et al for sample programs and more details:


REVISED
OpenCL 1.2
EDITION
Benedict R. Gaster Lee Howes
David R. Kaeli Perhaad Mistry Dana Schaa
Heterogeneous Computing
with OpenCL
MK

Field Programmable Gate Arrays

FPGA means *Field Programmable Gate Array*. This consists of a chip that can hold array of logical gates that can be programmed by the user. The chip can be programmed to perform very specialised tasks.

- For example, you could have a random number generator in a chip that can only generate random numbers.
- If done efficiently this can result in substantial acceleration of execution.
- As usual management of transfer latency is key.

Some banks are actively using FPGAs to risk manage their books (or maybe spreading rumours that they are so as to confuse their competitors).

Field Programmable Gate Arrays

Programming FPGAs

FPGAs are programmed in special languages called Hardware Definition Languages (HDL). The two main strands of HDL used are:

1. VHDL. This stands for VHSIC Hardware Definition Language and VHSIC stands for Very High Speed Integrated Circuits.
2. Verilog.

Note that programming an FPGA involves reducing all the program to bits and logical operations between bits. You have no floating point numbers or any numbers, you need to code these yourself.

Specialised software

KdB

- KdB is a database that is highly performance optimised for use with very large datasets that are managed in-memory.
- It is produced by a company called Kx (`kx.com`)
- The database is managed with a language called `K` which which fits in the L1 cache of the CPU and therefore runs very fast.
- KdB is used a lot by hedge funds in High Frequency Trading.
- The `K` language is derived from the language `APL` (which stands for *A Programming Language*). The functionality is usually accessed with a "simpler" language called `Q`.

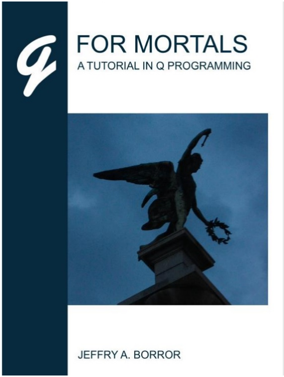
Specialised software

KdB

- Both `K` and `Q` are relatively complex languages:

Language	Code	Description
`Q`	`{$[x=0;1;x*.z.s[x-1]]}`	Calculate *x* factorial
`K`	`(!R)&{&/x!/:2_!x}'!R`	List all primes below *R*

- Professional `KdB/K/Q` specialists are very sought after and well paid by hedge funds and banks.
- You can use the 32 bit version for free: `http://kx.com/software-download.php`
- A reference is *Q for mortals* by Jeffry Borror



q FOR MORTALS
A TUTORIAL IN Q PROGRAMMING
JEFFRY A. BORROR