Making your programs faster

What I learnt on a 3 day course at Daresbury...

Roger Barlow

Int. Inst. for Accelerator Applications
The University of Huddersfield

June 8, 2016

Contents

- Basics
- Vectorisation
  - SIMD Architectures
  - How to invoke vectorisation
  - How to exploit vectorisation: peel, remainder, and stride
- Latency
- Profiling VTUNE or Amplifier
- Concurrency
  - Cores and threads
  - OpenMP and MPI
  - How-to with OpenMP
  - How-to with MPI

Basics

Standing instruction (1). Don't bother.

Who cares whether your program runs in 0.23 seconds instead of 3.2 seconds? Todays's computers are fast!

99% of the time this is true. But suppose we are in the other 1%.....

Standing instruction (2). Common sense.

Many computations produce results that are never used (at some level). Let's assume that you've cut all those out.

Standing instruction (3). No I/O

Printing, plotting, or file read/writes inside a loop will really slow it down .

Reduce the number of machine code instructions

Code like

`for (i=0;i<100; i++) {x[i]=sqrt(2)*y[i];}`

evaluates $\sqrt{2}$ 100 times. Better to write

`float rt2=sqrt(2); for (i=0;i<100; i++) {x[i]=rt2*y[i];}`

Unroll (small) loops:

```
    for(i=0;i<3;i++){x[i]=y[i]+z[i];}
    → x[0]=y[0]+z[0]; x[1]=y[1]+z[1]; x[2]=y[2]+z[2];
```

Use registers efficiently;

```
    x=y*z; a=b*c; u=v*x; → x=y*z; u=v*x; a=b*c;
```

Use inline functions.

Avoid unnecessary integer/float/double type conversions:

```
    float diam=2*r; → float diam=2.0*r;
```

Standing instruction (4). The Compiler is good at this - probably better than you are

Specify Optimisation level `gcc -o prog.X -O`*n* `prog.cc`

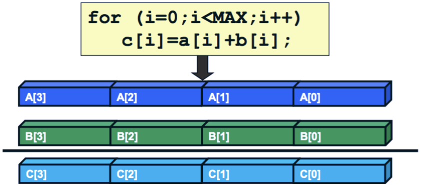
where n =0,1,2,3 increases optimisation

May be risky! Check test cases, especially if using level 3

Vectorisation

Also known as SIMD - Single Instruction Multiple Data

Some processors have vector arithmetic units. These are registers that, instead of the usual 8 bytes, contain 16 (SSE), 32 (AVX, AVX2), or even 64 bytes (AVX-512).
So they can handle 2,4 or even 8 doubles (or 4, 8 or even 16 floats/integers) at once. (And even more short integers)
(They appear in assembly listings with names starting x, y or z. There are also special instructions that manipulate them efficiently.)



```
for (i=0;i<MAX;i++)
   c[i]=a[i]+b[i];
```

How to invoke vector arithmetic

You don't have to - the compiler does it automatically! (Option -no-vec turns it off)
So `for (i=0;i<16;i++) { p[i]=q[i]*r[i];}` can be done in one go
Use -x*processor* to compile using special features, e.g. -xAVX
or -xHOST unless you are cross-compiling
How can you tell? -qopt-report*level* *level* goes from 1 to 5
icc -c -qopt-report5 -qopt-annotate myprog.c

Tweaking Vectorisation(1)

Loops cannot be vectorised if the order of iteration matters:
`for(i=0;i<100;i++) x[i]=3*x[i];` YES
`for(i=1;i<100;i++) x[i]=3*x[i-1];` NO
`for(i=0;i<100;i++) x[i]=3*y[i];` Unknown Looks OK, but what if x and y arrays overlap? `float* y=&x-1;`
Can be ambiguous if this is inside a function and arrays passed as arguments;
Use `#pragma` statements to tell the compiler it's OK to vectorise.
`#pragma noalias (x,y)`
#pragma ivdep - assures there are no dependencies (may be subtle if arrays and/or indices passed as function arguments)
#pragma simd - insists

Tweaking Vectorisation (2)

Vector operations work only for sets of numbers aligned on the relevant boundary, and only on complete vectors
So if `float p[10]` starts at word address xA0001, a loop with AVX2 vectorisation will
i) deal with p[0],p[1], and p[2] (three instances). This is called the 'peel'
ii) deal with p[3] to p[6] in one vectorised instance iii) deal with p[7], p[8], and p[9] (three instances) This is called the 'tail'
So:
1) Choose array dimensions carefully. If necessary add dummies at the end
2) Force alignment on the correct vector boundary. Use `_aligned_malloc`
3) Tell the compiler the variables are aligned

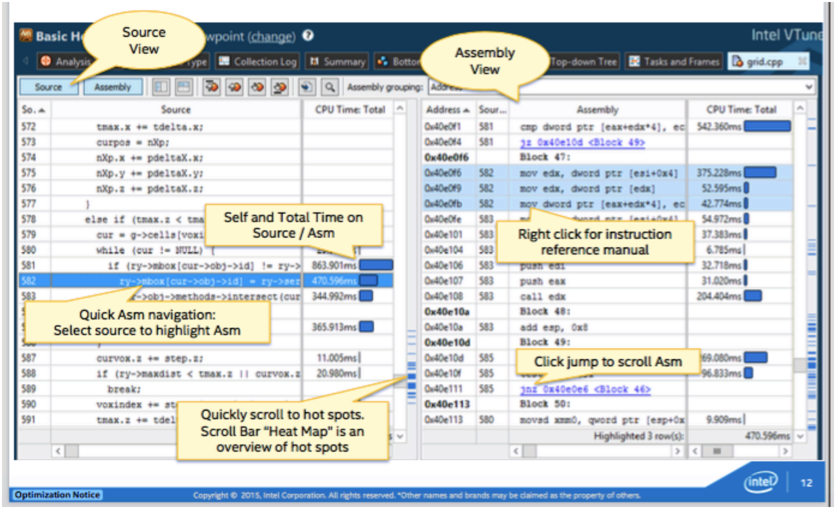
Latency

Not all instructions are equal!
Register/register instruction typically one clock cycle
Memory access may take several cycles. Some (local) memory is faster.
Cache system maps blocks into fast local memory.
(1) Tune cache parameters
(2) Worry about the way your arrays map onto memory
Key performance indicator is the CPI - Cycles per instruction

Profiling

Don't sweat the small stuff

Where in your code is the computer spending it's time? (Hotspots)
Profile with Intel VTune/Amplifier


Basic H[illegible] wpoint (change)
Intel VTune
Analysis
Collection Log
Summary
Bottom
Top-down Tree
Tasks and Frames
grid.cpp
Source
Assembly
Assembly grouping: Address
Source View
Assembly View
Self and Total Time on Source / Asm
Right click for instruction reference manual
Quick Asm navigation: Select source to highlight Asm
Click jump to scroll Asm
Quickly scroll to hot spots. Scroll Bar "Heat Map" is an overview of hot spots
Optimization Notice
intel
12

So.	Source	CPU Time: Total
572	`tmax.x += tdelta.x;`	
573	`curpos = nXp;`	
574	`nXp.x += pdeltaX.x;`	
575	`nXp.y += pdeltaX.y;`	
576	`nXp.z += pdeltaX.z;`	
577	`}`	
578	`else if (tmax.z < tma`	
579	`cur = g->cells[voxi`	
580	`while (cur != NULL) {`	[illegible]
581	`if (ry->mbox[cur->obj->id] != ry->`	863.901ms
582	`ry->mbox[cur->obj->id] = ry->ser`	470.596ms
583	`...->obj->methods->intersect(cur`	344.992ms
[illegible]	[illegible]	
[illegible]	[illegible]	365.913ms
[illegible]	`}`	
587	`curvox.z += step.z;`	11.005ms
588	`if (ry->maxdist < tmax.z || curvox.z`	20.980ms
589	`break;`	
590	`voxindex += st`	
591	`tmax.z += tdel`	

Address	Sour...	Assembly	CPU Time: Total
0x40e0f1	581	`cmp dword ptr [eax+edx*4], ec`	542.360ms
0x40e0f4	581	`jz 0x40e10d <Block 49>`	
0x40e0f6		`Block 47:`	
0x40e0f6	582	`mov edx, dword ptr [esi+0x4]`	375.228ms
0x40e0f9	582	`mov edx, dword ptr [edx]`	52.595ms
0x40e0fb	582	`mov dword ptr [eax+edx*4], ec`	42.774ms
0x40e0fe	583	`m[illegible] dword ptr [esi+0x4]`	54.972ms
0x40e101	583	[illegible]	37.383ms
0x40e104	583	[illegible]	6.785ms
0x40e106	583	`push edi`	32.718ms
0x40e107	583	`push eax`	31.020ms
0x40e108	583	`call edx`	204.404ms
0x40e10a		`Block 48:`	
0x40e10a	583	`add esp, 0x8`	
0x40e10d		`Block 49:`	
0x40e10d	585	[illegible]	69.080ms
0x40e10f	585	[illegible]	96.833ms
0x40e111	585	`jnz 0x40e0e6 <Block 46>`	
0x40e113		`Block 50:`	
0x40e113	580	`movsd xmm0, qword ptr [esp+0x`	9.909ms
		Highlighted 3 row(s):	470.596ms

Cores and threads

Cores are hardware. A core is essentially an independent CPU. It has its own registers and program counter. Machine code instructions are loaded, decoded and acted on (using pipelining, so it's complicated)

Typical i5 PC has 4 cores. Server like IIAA1 has 24. Xeon Phi has 60. If you're not using all the cores, you're inefficient.

Threads are software. A typical program is one thread (do this, then do this, then do this...) but with care can be split into more than one.

A computer will be running with lots of threads active. User(s) plus system processes. Do `top` to see some of them, or `ps -e`

Any given core runs a particular thread. So only *ncores* of the active threads are actually running. It will switch to another thread if (i) this thread is waiting for something or (ii) the scheduler tells it to. Switching takes time (to store thread registers and load new ones)

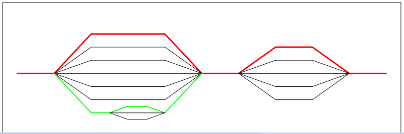
Hyperthreading is an exception. Cores with hyperthreading can switch seamlessly between two threads. So an i7 CPU with 4 cores can have 8 threads running, but each is, on average, only using half the clock cycles.

OpenMP and MPI

Two different systems with deceptively similar names -

MPI - Message Passing Interconnect is basically designed for distributed systems doing their own thing but talking to each other

OpenMP - Open Multi-Processing is basically designed for shared-memory systems. Executes the same block of code in several parallel threads.



OpenMP

Very easy to use:
1) Include header file `<omp.h>`
2) Compile using `-fopenmp` flag
3) Use `#pragma omp` directives to fork process into threads
Each of the produced theads has a number accessible through `omp_get_thread_num()`. Thread 0 is the 'master' thread, others are 'slaves'.

```
#include <iostream>
#include <omp.h>
void main(){
using namespace std;
#pragma omp parallel
{int i=omp_get_thread_num();
int n=omp_get_num_threads();
cout<<"Hello World from " <<i<<" of "<<n<<endl;
} }
```

What happens...

```
$ export OMP_NUM_THREADS=6
$ ./a.out
Hello World from Hello World from Hello World from 52 of 6
of 61 of 6Hello World from Hello World from
Hello World from 3
4 of 6
0 of 6
of 6
```

Other useful OpenMP stuff

Default is numthreads = numprocessors. Override `export OMP_NUM_THREADS=`*n*

#pragma directives apply only to the next item. Use {} to create a block.

cout not 'thread-safe'. Output may be all mixed up. Can safeguard by #pragma omp critical
Generally used to avoid *race condition*

#pragma omp master means only the master thread will run this
#pragma omp single means only one thread will run this

#pragma omp barrier means wait until all other threads have caught up. Implicit barrier at the end of the parallel block - can remove with `nowait`

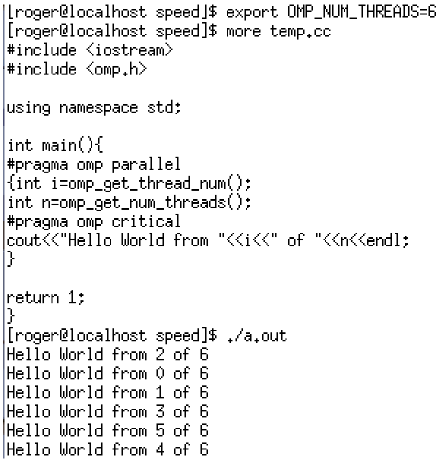
Variables declared inside the pragma block are private, but those declared outside are shared. Update at your peril! Can override.

Also very useful `#pragma omp parallel for` followed by for loop.

`#pragma omp parallel` can be nested

Setting the AFFINITY environment variable can give a big speedup

OpenMP - spot the difference



```
[roger@localhost speed]$ export OMP_NUM_THREADS=6
[roger@localhost speed]$ more temp.cc
#include <iostream>
#include <omp.h>

using namespace std;

int main(){
#pragma omp parallel
{int i=omp_get_thread_num();
int n=omp_get_num_threads();
#pragma omp critical
cout<<"Hello World from "<<i<<" of "<<n<<endl;
}

return 1;
}
[roger@localhost speed]$ ./a.out
Hello World from 2 of 6
Hello World from 0 of 6
Hello World from 1 of 6
Hello World from 3 of 6
Hello World from 5 of 6
Hello World from 4 of 6
```

MPI - toy program

4 of the 6 essential MPI functions

include mpi.h header file
compile with mpicc -o program program.cc
run with mpirun -np *nprocs* program

```
#include <mpi.h>
#include <iostream>
void main(){
MPI_Init();
int rank,size;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
cout<<" process "<<rank <<" out of "<<size <<endl;
MPI_Finalize();
}
```

So far not much different from OpenMP - but all processes are started together, and variables private

MPI - doing stuff

The other two functions

`MPI_send(buffer,size,datatype,dest,tag,MPI_COMM_WORLD)`
Sends a buffer to process `dest`. `datatype` is `MPI::INT`, `MPI::FLOAT` or whatever. `tag` is for your convenience .
This returns only when the message has been safely sent ('blocking'). Nonblocking alternatives are available

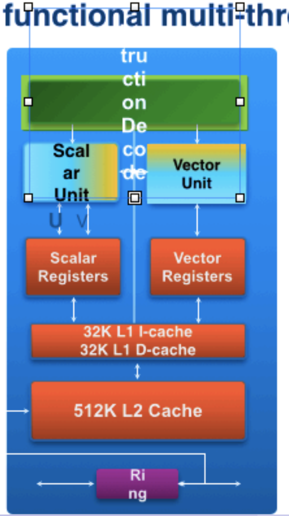
`MPI_receive(buffer,size,datatype,source,tag,MPI_COMM_WORLD,& status)`
Waits for any message of type `tag` from process `source` of this `datatype`. `size` is the maximum - to find the actual size use
`MPI_Get_count(&status,datatype,&count)`

So a typical program has a switch on the rank, and undertakes different roles accordingly. The rank 0 (master) process will send messages and wait for answers, the others will wait for messages and send answers.

the Xeon Phi

Slide stolen from Steve Pickles

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



Instruction decoder is fully pipelined but is designed as a 2-cycle unit

- Enables significant increase to maximum core frequency, but...
  - Core cannot issue instructions from same context in adjacent cycles
  - **Means minimum two threads per core to use all available compute cycles**

How to use them

Phi behaves like separate computer - can log in to it from the host with

```
ssh mic0
```

(lots of this Phi stuff is called 'mic' - for Many Integrated Core architecture
Host and Phi share filespace
Compile Phi programs on the host with -mmic option. (Host and Phi binaries are not compatible)

Or can run in the host and offload to the Xeon Phi - as a co-processor

```
#pragma offload target(mic) input(A:length(2000))
```

Or can run openmp and/or mpi
or various combinations of these: offloaded block can contain openmp parallelism to exploit all the cores
The Intel Math Kernel Library (MKL) can take full advantage. Automatic offload by setting `MKL_MIC_ENABLE=1`

Plans.....

We have 4 Xeon Phis each wth 60 cores!

And we're not using them

M Eng project student didn't manage - but we learned a lot.

1. Back up user files, including software (Geant4, talys...)
2. Replace Fedora23 with CentOS
3. Restore user files (should be no difference - except your password will change)
4. Re-install various packages (torque, httpd....)
5. Install Intel MPSS, MKL, compilers etc
6. Start exploiting all those cores!