Durham University
Department of Computer Science

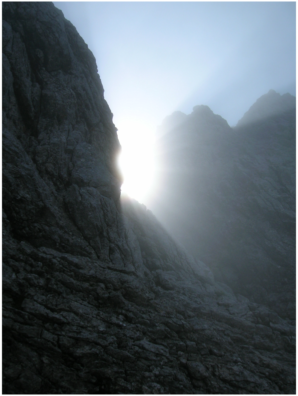
Core I—Introduction to HPC

Session III: Vectorisation

Dr. Weinzierl

Michaelmas term 2018

Outline



Parallelisation concepts
Autovectorisation and compiler feedback
Manual vectorisation

Memory architectures

Durham University
Department of Computer Science

We use two types of machines in this lecture:

Shared Memory Architecture

Communication:

- one core places value in memory
- next core reads it

Memory access conflicts:

- Read-write
- Write-read
- Write-write

We may only read concurrently

Distributed Memory Architecture

Communication:

- one core sends out message
- next core receives it

Algorithmic challenges:

- Deadlock
- Data consistency

Distributed shared memory:

- Pretend that the memory is shared though it is not
- Can be simulated by intermediate software layer
- Can be offered through remote data access

Programming model

Durham University
Department of Computer Science

Annotations

- Take serial source code
- Insert new comment-like keywords
- Precompiler/source-to-source compiler

Explicit API calls

- Take serial source code
- Use new functions
- Link with library

All approaches realise *explicit parallelism* (not automatically done by compiler).
Auto-parallelism would be the prime solution!

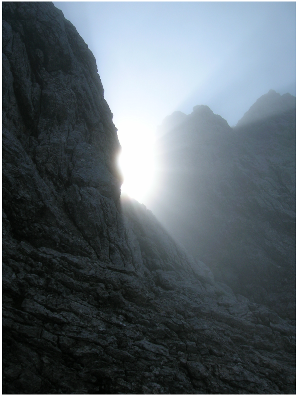
Concept of building block

Durham University
Department of Computer Science

- Content
  - Introduce shared vs. distributed memory machines
  - Discuss appropriate programming models
- Expected Learning Outcomes
  - The student can describe differences of the machines
  - The student can classify new programming techniques/models

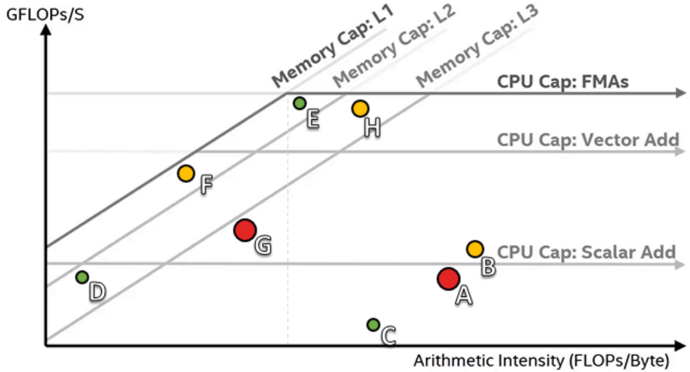
Outline

Durham University
Department of Computer Science



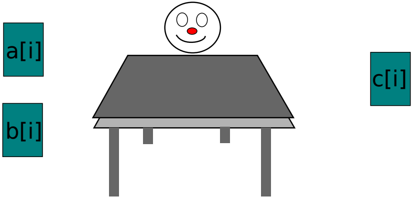
Parallelisation concepts
Autovectorisation and compiler feedback
Manual vectorisation

Insights from the roofline model (recap)



- A Not bandwidth-bound, would benefit from vectorisation
- B Insufficient vectorisation
- D L2 cache-bound
- G Memory bandwidth-bound

SISD (recap)



$c = a + b \qquad a, b, c \in \mathbb{R}^N$

- SISD architecture: One ALU and one activity a time
- Runtime: $(2 \cdot \textit{load} + \textit{add} + \textit{store}) \cdot N$
- Usually *load*, *add*, *store* require multiple cycles (depend both on hardware and environment such as caches; cf. later sessions)

Pseudoassembler code with SISD (recap)

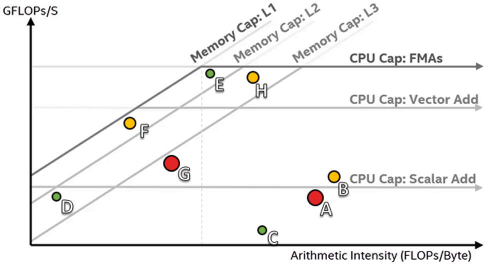
Durham University
Department of Computer Science

$$c = a + b \qquad\qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N; i++) {
  load a[i]
  load b[i]
  add
  store c[i]
}
```

Reading roofline plots

Durham University
Department of Computer Science



With a roofline model and a code analysis we can

- predict impact of vectorisation
  (move dot up)
- clarify whether vectorisation is worth the effort
- identify code parts which require algorithmic changes
  (move dot to the right)

Agenda

Durham University
Department of Computer Science

Compilers today do quite a lot of vectorisation automatically, but

- we have to understand where they succeed and where not
- we have to help them from time to time
- we have to write down code in a proper way

Vectorisable loops

Durham University
Department of Computer Science

Loops are vectorisation candidates iff

1. Countable (cmp. C for to for-loops in theoretical computer science)
2. Single entry, single exit
3. No branches
4. Only inner loops
5. No function calls (cmp. inlining)
6. No internal dependencies

Most standard library (intrinsic) functions however are allowed as there are vectorised versions available.

Loops that can’t benefit from vectorisation

Durham University
Department of Computer Science

```
for (int i=1; i<N; i*=2) {
  if (x[i]<20.0) {
    y[i] = 0.0;
  }
  else {
    y[i] = x[i]/20.0;
  }
  foo( y[i] );
  y[i-1] = y[i] + 1.0;
}
```

Vectorisation reports (Intel)

Durham University
Department of Computer Science

Standard translation mode leaves us without a clue about whether the output will use vectorisation facilities, but we can ask the compiler to be verbose:

```
    [tobias@phi1 course-GPUProgramming]$ icpc -vec-report2 main.cpp
    ...
main.cpp(429): (col. 5) remark: loop was not vectorized: existence of vector dependence
main.cpp(428): (col. 3) remark: loop was not vectorized: not inner loop
main.cpp(106): (col. 3) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(107): (col. 5) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(108): (col. 7) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(118): (col. 3) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(119): (col. 5) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(120): (col. 7) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(135): (col. 3) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(136): (col. 5) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(137): (col. 7) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(146): (col. 3) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
main.cpp(147): (col. 5) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
```

- We rely on default optimisation level (`-O2`) here

Intel:

- Old style: `-vec-report2` (different levels available)
- Windows: `/Qopt-report:1 /Qopt-report-phase:vec`
- Linux: `-qopt-report=1 -qopt-report-phase=vec`

GNU:

- `-ftree-vectorize` switches on vectorisation
- Automatically triggered by `-O3`
- `-ftree-vectorize-verbose=N` yields reports

Countable loops

Countable loop: The number of iterations is known (and fixed) when we enter the loop.

```
for (int i=0; i<N; i++) {
  if (b[i]<1.5) {
    break;
  }
  c[i] = a[i] * b[i];
}
```

Translates with `-vec-report2`:

```
myfile.cpp(125): remark: loop was not vectorized: nonstandard loop
is not a vectorization candidate.
```

Ideas:

- Can you predict a priori whether break holds?
- Can you roll-back?
- Can you mask results?

Loops and their counters

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

- Access though array index preferable to pointer arithmetics.
- Loop counter modification in loop body problematic.

Successful vectorisation:

```
myfile.cpp(125): remark: vectorization support: unroll factor
                         set to 4.
myfile.cpp(125): remark: LOOP WAS VECTORIZED.
```

Behind the scenes: unroll factors

Compilers can unroll simple loops even though the looper counter is not known a priori:

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

Unrolled code fragment for 4-way SIMD instructions (pseudo code):

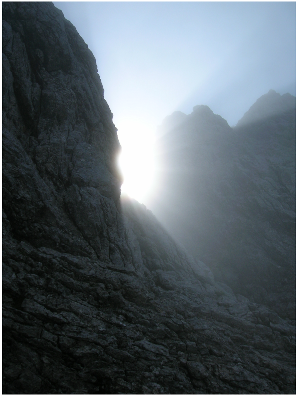
```
for (int i=0; i<N/4*4; i+=4) {
  c[i+0] = a[i+0] * b[i+0];
  c[i+1] = a[i+1] * b[i+1];
  c[i+2] = a[i+2] * b[i+2];
  c[i+3] = a[i+3] * b[i+3];
}
for (int i=N/4*4; i<N; i++) {
  c[i] = a[i] * b[i];
}
```

Concept of building block

Durham University
Department of Computer Science

- Content
  - Reiterte message behind roofline plots
  - Discuss compiler feedback
  - Discuss mandatory loop properties for vectorisation
- Expected Learning Outcomes
  - The student can explain roofline models
  - The student can interpret compiler feedback
  - The student can explain why certain code snippets vectorise or do not vectorise

Outline



Parallelisation concepts
Autovectorisation and compiler feedback
Manual vectorisation

Vectorisation paradigms

Durham University
Department of Computer Science

- Automatic (compiler)
  - Rely on compiler only to find vectorisable code fragments
  - CUDA, e.g., pushes this idea
- Assembler
  - Manually load, store data and trigger right calls
  - Portability (Xeon Phi's AVX, e.g., is not Sandy Bridge SSE/AVX)
- Intrinsics
  - Special typedefs and functions provided in header file (library approach)
  - Replace code fragments by function calls (inlining assembler code)
- Pragmas
  - Augment source code with compiler hints
  - It is up to the compiler to find hardware-specific solution
- Domain-specific languages or language extensions

Why manual vectorisation is tricky

Durham University
Department of Computer Science

- Hardware changes rapidely
  - Sandy Bridge: 256-bit FMUL and 256-bit FADD per core (DP: 16 values held in registers; 8 Flops/cycle)
  - Haswell: Two 256-bit FMA per core (DP: 16 Flops/cycle)
- MIC
  - Four threads share one FPU
  - Scheduling is non-trivial
- Latency
  - Theoretical flops/cycle are upper bound
  - Vectorisation often does not pay off due to latency constraints
- The API (intrinsics) changes
- to be continued

$\Rightarrow$ We will rely on the compiler to vectorise in this course!

Pragmas in C

Durham University
Department of Computer Science

The `#pragma` directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. Three forms of this directive (commonly known as pragmas) are specified by the 1999 C standard. A C compiler is free to attach any meaning it likes to other pragmas.

Pragma example: inlining

> Copy 'n' paste function body into calling source code and thus avoid function call overhead.

```
for (int i=0; i<N; i++) {
  c[i] = foo(i/4);
}
```

If `foo` is evaluated per loop iteration, we might end up spending all of our precious cycles in call stack administration (call overhead).

- Option 1: `inline` keyword in C (does work with `ipo` if `ipo` works)
- Option 2: Manual inlining by placing `foo` into header files
- Option 3: Manual inlining by copy 'n' paste yourself
- Option 4: Enforce inlining with pragmas

Inlining with pragmas

Durham University
Department of Computer Science

```
for (int i=0; i<N; i++) {
  #pragma forceinline recursive
  c[i] = foo(i/4);
}
```

- `#pragma inline` Hint to the compiler
- `#pragma inline recursive` Hint to the compiler to inline recursively
- `#pragma forceinline` Forces compiler to inline this particular function but not functions called within this function
- `#pragma forceinline recursive` Forces compiler to inline recursively (complete unroll of call graph)
- `#pragma noinline` Avoid inlining and reduce code size.
- These are Intel options. GCC requires you to annotate functions (`__attribute__((always_inline))`)

> Compiler inlining: If we use `-O3`, the compiler automatically inlines many small helper functions (cmp. outcome of profiling in second session). Which routines are inlined however depends on a (unknown) heuristic whereas an explicit inline call enforces the compiler to inline.

Enforced unrolling (Intel)


```
#pragma unroll (5)
for (int i=0; i<N; i++) {
  c[i] = a[i] * b[i];
}
```

- Recommends the compiler to unroll loop even though its internal heuristics recommend not to do so (instruction latency).
- If unroll factor is omitted, optimiser tries to determine/predict good unroll factor.
- Usually only recommended, if vectorisation report tells you that loop is not unrolled though you know that loop body is expensive (due to inlined function; cmp. follow-up discussion).
- Helps you to unroll outer loops and thus enable further optimisations; for inner loops usually useless.

```
#pragma nounroll
for (int i=0; i<N; i++) {
  c[i] = a[i] * b[i];
}
```

Branches and vectorisation

Durham University
Department of Computer Science

```
for (int i=0; i<N; i++) {
  if (a[i]>=0.0) {
    c[i] = a[i] * b[i];
  }
  else {
    c[i] = a[i] + b[i];
  }
}
```

Handbooks suggest that this code does not vectorise, however:

```
myfile.cpp(125): remark: vectorization support: unroll factor set to 4.
myfile.cpp(125): remark: LOOP WAS VECTORIZED.
```

Both variants are evaluated, but:

Masking Result of computation is either used or discarded.

Blending Two results are computed but one is discarded.

Source code dependencies

Durham University
Department of Computer Science

```
c[i]   = a[i-1]*3;
c[i+1] = a[i+1-1]*3;

c[i]   = c[i-1]*3;
c[i+1] = c[i+1-1]*3;
```

- Dependencies between computations imply that the code fragment cannot be vectorised.
- Dependency types are:
  - Read after read (no conflict)
  - Write after read
  - Read after write
  - Write after write (semantics?)
- `-vec-report6` plots dependencies

Durham University
Department of Computer Science

Dependency analysis: read after write

```
for (int i=1; i<N; i++) {
  myA[i] = myA[i-1] * myB[i];
}
```

```
myfile.cpp(125): remark: loop was not vectorized: existence of vector
dependencies.
myfile.cpp(125): remark: vector dependence: assumed FLOW dependence
between myA line 125 and myA line 125.
```

You can make the compiler ignore this dependency due to

```
#pragma ivdep
```

Further pragmas

- `#pragma loop_count(10)` Tell the compiler about loop count and thus guide heuristics.
- `#pragma vector always` Force compiler to vectorise even though heuristics indicate that it is inefficient.
- `#pragma simd` Enforce vectorisation.

`#pragma simd` includes `ivdep` and thus might lead to broken code.

Summary:

- In the end, it is often trial and error.
- Vectorisation reports however guide you through process.
- Often, it does not require major code changes but gives nevertheless that whoa effect.

Concept of building block

Durham University
Department of Computer Science

- Content
  - Reiterte message behind roofline plots
  - Discuss compiler feedback
  - Discuss mandatory loop properties for vectorisation
- Expected Learning Outcomes
  - The student can explain roofline models
  - The student can interpret compiler feedback
  - The student can explain why certain code snippets vectorise or do not vectorise