SICM^2 Software Summer School: Parallel algorithms and programming models

Robert Harrison[1]

[1] Stony Brook University

July 2014
Stony Brook, NY

High-performance computing — objectives

- Do science using the computer
  - Emphasizes productivity, speed of testing new ideas, correctness
- Science is collaborative, funding limited, software long lived
  - Need readable, extensible, maintainable, portable software
- Science is done by science students not computer scientists
  - Ideally code in high-level concepts, hiding arcane computer details
- Big/many calculations and/or a large user community
  - Demand high-performance, efficient use of resources, robustness

Practical high-performance computing

- Performance is a level-1 correctness issue
- How to make a computer run fast?
- How to do this portably?
  - Current machines
  - Future machines
- How to do this while maintaining productivity?

Technology trends

- Power wall
  - Computer clock frequencies are no longer steadily increasing
- Memory wall
  - Memory, communication, and disk bandwidth increasing less rapidly than compute speed
  - New technologies (e.g., stacked memory) essential to progress
- Parallelism is the **only** path to increased performance
  - Fine-grain
    - instructions — MIMD
    - vectors — SIMD
  - Medium-grain
    - cores — MIMD
  - Coarse-grain
    - sockets, nodes — MIMD
- Good news — technology is delivering more fine/medium grain
  - Phew! 1M-way coarse grain parallelism is too much!
  - What to do with all those transitors?

Connecting concepts in architecture, software, and tools

Architectural element	**Software construct**	**Programming tool**
within a "core"		
pipe-lined units	complex expressions, loops	compiler, library
multiple instruction	complex expresions, fat loops	compiler
simd units	loops, ops on vectors/matrices	compiler, library
within a "node"		
cores/thread units	thread/process/task	openmp, pthreads, tbb
	loop nests, ops on matrices	openmp, opencl, libraries
accelerators	task, ops on matrices	opencl, openacc, cuda, libs
between "nodes"		
distributed memory	outer loops, multiple tasks, big matrices	mpi, libraries

Just do it!

- Overthinking all this is counter productive
- Write pipelineable, vectorizable code — more on this later
- Have the compiler do the work

```
for (int i=0; i<n; i++) {
   c[i] = a[i]*23.0 + exp(d[i]*d[i]);
}
```

Hello world vector style

- Code

```
#include <iostream>

int main() {
    double a[100];
    for (int i=0; i<100; i++) a[i] = i;
    double sum = 0.0;
    for (int i=0; i<100; i++) sum += a[i];
    std::cout << sum << std::endl;
    return 0;
}
```

- Compilation

```
$ icpc -fast -vec_report3 sum.cc
sum.cc(5): (col. 5) remark: FUSED LOOP WAS VECTORIZED.
```

- Output ($\sum_{i=0}^{n-1} i = n(n-1)/2 = 4950$)

```
$ ./a.out
4950
```

Thinking about performance

- What is the bottleneck
  - Data motion?
  - Instruction issue?
  - Functional units?
- Moving data
  - disk
  - main memory
  - some level of cache
  - registers
- Performance model

$$t(n) = L + \frac{n}{B}$$

L = latency, B = bandwidth, n = number of elements

Hockney's $n_{1/2}$

- How long must a loop be, or how much data must we move, in order to hit 50% performance

$$\text{rate}(n) = \frac{n}{t(n)} = \frac{n}{L + \frac{n}{B}} = \frac{B}{\frac{BL}{n} + 1}$$

- The asymptotic rate is just

$$\text{rate}(\infty) = B$$

- Hence,

$$\text{rate}(n_{1/2}) = \frac{B}{2} \rightarrow n_{1/2} = BL$$

- also

$$\text{rate}(n_{90\%}) = 0.9B \rightarrow n_{90\%} = 9BL$$

E.g., typical message passing between processes

- latency 5us, bandwidth 1e9 bytes/s
- $n_{1/2} = 1{,}000$ bytes
- $n_{90\%} = 9{,}000$ bytes

Pipelined functional units

- Parallelism arises from overlapping stages of performing successive operations
- E.g., floating point a*b
  - A single operation may take 3 cycles to complete (pipeline depth or latency)
  - You can issue a new operation every cycle

cycle	stage0	stage1	stage2	result
1	a0*b0			
2	a1*b1	a0*b0		
3	a2*b2	a1*b1	a0*b0	
4	a3*b3	a2*b2	a1*b1	a0*b0
5	a4*b4	a3*b3	a2*b2	a1*b1
⋮	...			

Pipelined functional units

- Serial execution
  - Wait for each result to complete
  - The cost per operation is the pipeline depth
- Pipelined execution
  - $t(n) = L + n - 1$ (in cycles)
  - $n_{1/2} = L - 1$
  - $n_{90\%} = 9 * (L - 1)$ 9*2 = 18

Multiple instruction issue

- High-end x86 can issue each cycle at least
  - One or more integer operations (e.g. inc loop counter)
  - A test+branch operation
  - A (simd) floating point multiply
  - A (simd) floating point addition
  - Two (simd) memory operations
- The integer units typically have 1 cycle latency
- FMA — fused multiply-add — `a*b+c-->d`
  - Introduced with Haswell (`http://goo.gl/Bo3Rt`, `http://goo.gl/tePyD`)
- The compiler does this for you — usually better than humans except on very tight loops.

SIMD (vector) operations

- Single instruction applying operation to multiple data

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} * \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \rightarrow \begin{pmatrix} a_0 * b_0 \\ a_1 * b_1 \\ a_2 * b_2 \\ a_3 * b_3 \end{pmatrix}$$

- Older x86 — SSE — 2 doubles (128 bit)
- Newest x86 — AVX(2) — 4 doubles (256 bit), predicates, special functions, gather, scatter
- MIC x86 — LRBNI — 8 doubles (512bit), predicates, special functions, gather, scatter
- Zillions of different operations (`http://goo.gl/D0d7t`)
- Operations are pipelined
- $n_{90\%} = 9 * W * (L - 1) = 108$ with $W = 4$ (width) and $L = 4$ (latency)

Serial scalar v.s. pipelined SIMD performance

- Serial scalar computation produces 1 result every L cycles
- Pipelined SIMD produces W results every cycle
- Pipelined is WL times faster which is 16 with $W = L =4$
- Allowing for 2 operations per cycle (*,+) this is becomes 32
- I.e., worst case serial, non-vector code can be 32x slower!!!!
- Whenever some twit talks about GPGPUs being 100+ times faster than a CPU ask about how well vectorized (and threaded) the CPU code was.

Example performance — DAXPY

- Measure cycles/iteration of this loop as a function of n

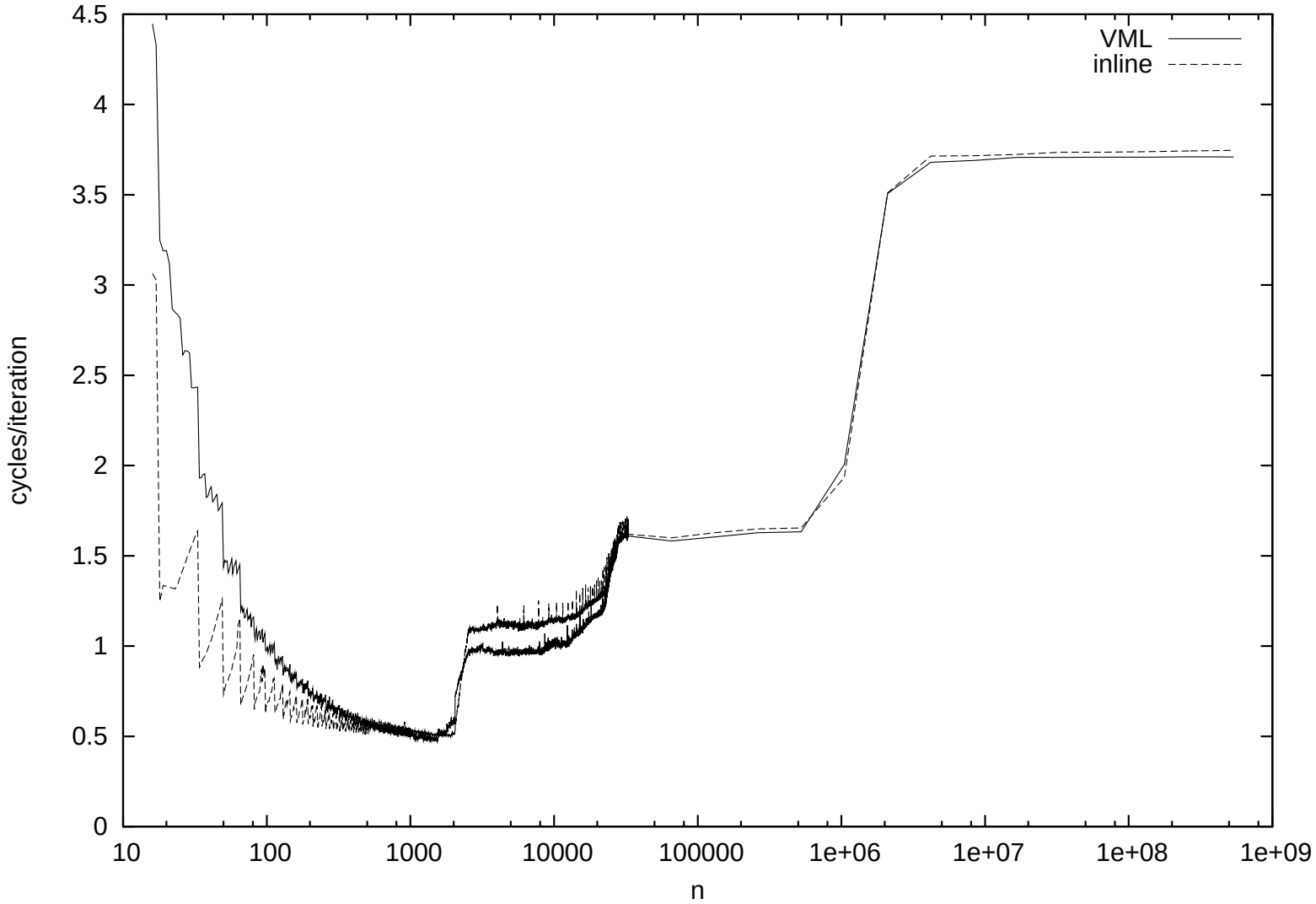
```
for (int i=0; i<n; i++) {
    y[i] = a*x[i] + y[i];
}
```

or from VML

```
cblas_daxpy (n, a, x, 1, y, 1);
```

- What are you expecting to see?

Example performance — DAXPY



DAXPY performance analysis

- In two cyles can do 2 loads, *, +, store, increment, branch
- AVX SIMD is 4 wide
- Hence, 4 iterations per 2 cycles = 0.5 cycles/iter at best
- L1 = 32KB (4096 dbl), 48 bytes/cycle
  (capable of 64 bytes/cycle?)
- L2 = 256KB (32768 dbl), 21 bytes/cycle
- L3 = 8MB (1M dbl), 14.8 bytes/cycle
- main memory, 6.4 bytes/cycle

DAXPY performance analysis

- Lesson — cache fast, memory slow (latency even more so)
- Assignment — predict then measure performance of

```
for (int i=0; i<n; i+=4) {
    y[i] = a*x[i] + y[i];
}
```

  (remember to account for reduced number of elements in perf. anal.)
- Assignment — what if stride is 4096 instead of 4?

Read the documentation — really!

- The tools do a **lot** more than most people realize.
- Will introduce lots of valuable concepts and techniques.
- Read the release notes with every new version.
- Intel developer zone `http://software.intel.com/`
- Intel compilers `http://software.intel.com/en-us/intel-compilers`
- Intel composer software suite

  `http://software.intel.com/en-us/articles/intel-c-composer-xe-2011-documentation`
- Intel C++ compiler `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm`
- Getting started with auto-vectorization `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/tutorials/lin/cmp_vec_c/index.htm`

More Intel links

- Go parallel portal `http://go-parallel.com/`
- Learning lab `http://software.intel.com/en-us/intel-learning-lab/`
- Vectorization `http://software.intel.com/en-us/intel-vectorization-tools/`
- Evaulation guide portale `http://software.intel.com/en-us/evaluation-guides/`
- Parallel magazine `http://software.intel.com/en-us/intel-parallel-universe-magazine/`
- OpenCL `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`
- MxM using MKL `http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/tutorials/mkl_mmx_c/tutorial_mkl_mmx_c.pdf`

Non-vectorizable loops

- Dependencies between iterations

```
for (int=1; i<n; i++) a[i-1] = 3*a[i];
```

- Dependencies through memory

```
void f(int n double *a, double* b) {
  for (int=0; i<n; i++) a[i] = b[i];
}
```

- Dependencies through iteration index or count

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

- Calls to non-inline functions (math library exceptions)
- If-tests that don't resolve to vector merge

Passing your knowledge to the compiler

- `#pragma ivdep` — ignore vector dependencies
- `restrict`
- `-fargument-noalias` — function arguments cannot alias each other, but they can alias global storage
- `-fargument-noalias-global` — function arguments cannot alias each other or global storage
- `#pragma loop count (n)` — advises the compiler of the typical trip count of the loop.
- `#pragma vector always` — always vectorize if safe regardless of if performance improvement expected
- `#pragma vector align` — asserts that data within the following loop is aligned (to a 16 byte boundary, for SSE instruction sets)
- `#pragma novector` — asks the compiler not to vectorize a particular loop.

Passing your knowledge to the compiler

- `#pragma vector nontemporal` — gives a hint to the compiler that data will not be reused, and therefore to use streaming stores that bypass cache.
- `#pragma simd` — Lots of options. Forces vectorization even if it is unsafe.
- `__attribute__((vector))` and `__declspec(vector)` — Vectorization of functions when definition is not available for inlining.

Monte Carlo Example

- Metropolis Monte Carlo
  - General and powerful algorithm for multi-dimensional integration
  - Abuse it to create a simple test code that reflects real applications
- Compute average value of x sampled from the normalized probably distribution function e^{-x}, i.e.,

$$\langle x \rangle = \frac{\int_0^\infty x e^{-x} dx}{\int_0^\infty e^{-x} dx} = 1$$

Monte Carlo Example - II

- Starting from uniform random numbers in $[0, 1)$ use Metropolis algorithm to sample from e^{-x}
- Approxmate infinity as 23 ($exp(-23)$ =1e-10)
- Algorithm

```
x = 23.0*rand() // initialize
while (1) {
   xnew = 32.0*rand();
   if (exp(-xnew) > exp(-x)*rand()) x = xnew;
}
```

- Asymptotically, x is sampled from $exp(-x)$ (with some correlation between successive values)

Monte Carlo Example - II

- Intel(R) Xeon(R) CPU E5-2687W
  - `mc0` — 71.2 cycles/point
  - `mc1` — 71.8
  - `mc2` — 76.2
  - `mc3` — 54.3
  - `mc4` — 18.0
  - `mc5` — 14.8
- Intel(R) Xeon(R) CPU E5645 (Hokiespeed)

Variational Monte-Carlo Example

- Uses Metropolis algorithm to sample points distributed according to ψ^2 for the helium atom and the Hylleraas wave function

$$\psi(r_1, r_2, r_{12}) = (1 + \tfrac{1}{2} r_{12}) e^{-2(r_1 + r_2)}$$

- Variational since $\langle E_L \rangle >= E_0$ where $E_L = (\hat{H}\psi)/\psi$
- Computes $\langle r_1 \rangle$, $\langle r_2 \rangle$, $\langle r_{12} \rangle$

Assignments for vector programming

1. Read (at least skim) all the compiler reference manual
   Pay special attention to the autovectorization section
2. Work through the autovectorization getting started
3. Examine the model matrix multiplication code
4. Understand each of the Monte Carlo code versions and reproduce performance
5. Tune a kernel relevant to you (modify bench.cc to measure performance)

Shared-memory programming beyond data-parallel OpenMP

- OpenMP

  https://computing.llnl.gov/tutorials/openMP/

  https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

  http://www.cs.utah.edu/~mhall/cs4961f11/

  http://www.cs.utah.edu/~mhall/cs4961f11/CS4961-L9.pdf

- Pthreads

  https://computing.llnl.gov/tutorials/pthreads/

  http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

  http://cs.gmu.edu/~white/CS571/pthreadTutorial.pdf

- TBB

  http://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf

  http://stackoverflow.com/questions/13446434/tbb-beginner-tutorial

  http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm#

  reference/reference.htm

Spinlocks v.s. mutexes

- Spinlocks
  - Fastest with low contention and no false sharing
  - Low resource use (minimally one cache line)
  - Potentially very slow under contention, in virtual machines, oversubscribed
  - Spinning threads can saturate memory and slow everything
- Mutexes
  - Thread scheduler blocks waiting threads
  - Usually slower than spinlocks in low contention
  - Scale well under contention, in virtual machines, oversubscription
- Recommendations
  - Use trusted implementation of higher-level concept (e.g., thread-safe containers)
  - Think tasks, not threads
  - Avoid heavily contested critical sections
  - Use mutexes; try spinlocks if performance is poor

Critical section issues

- Fairness — not guaranteed but often assumed
- Correctly updating shared data structures is not easy
  - `volatile` — does not solve the underlying problem but is useful to get the compiler to help

    `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/faq.html`

    `http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming`

    `http://www.drdobbs.com/cpp/volatile-the-multithreaded-programmers-b/184403766`
  - Need memory fences and compile/run time barriers to instruction migration
  - Pthread routines provide these — home grown assembly probably does not
  - STL structures cannot be made volatile — but may not need to be

Other options

- Condition variables
- Fair, scalable synchronization
  `http://dl.acm.org/citation.cfm?id=103729`
- Again, think tasks not threads

Intel Thread Building Blocks