Supercomputing for Everyone Series: Performance Tuning Summer School

**L5: Core tuning pays off n-times**

Dr. Holger Brunst
Technische Universität Dresden, Germany

August 17–21, 2015
Organization of this class

- course runs August 17-21, from 11 a.m. until 4.30 p.m. (EDT)
- 20 sites, each site should have an instructor/TA/helper
- please put your microphones on mute, if you are not asking questions
- set up in lectures (60 min) and exercises (75 min)
- Q&A time at end of each lecture
- chat for discussions
- class material and links: [http://go.iu.edu/CtB](http://go.iu.edu/CtB)
Lunch breaks are important

Monday
11.00-12.30 L1_Performance is ambiguous
12.30-13.15 X1_First steps on Blue Waters
13.15-14.15 Lunch break
14.15-15.15 L2_Evaluation first!
15.15-16.30 X2_Pen, paper, and performance

Tuesday
11.00-12.00 L3_Use simple tools for simple questions
12.00-13.15 X3_The command line is your friend
13.15-14.15 Lunch break
14.15-15.15 L4_Tuning needs persistence
15.15-16.30 X4_Benchmarks provide the baseline

Wednesday
11.00-12.00 L5_Core tuning pays off n-times
12.00-13.15 X5_How to access data efficiently
13.15-14.15 Lunch break

Thursday
11.00-12.00 L7_Sharing may double the sorrow (OpenMP)
12.00-13.15 X7_OpenMP enables quick 'n easy gains
13.15-14.15 Lunch break
14.15-15.15 L8_Hand made parallelization hurts (MPI)
15.15-16.30 X8_From bad MPI to good MPI

Friday
11.00-12.00 L9_Climb the mount Olympus with GPUs
12.00-13.15 X9_Hybrid tuning in practice
13.15-14.15 Lunch break
14.15-15.15 L10_The grand final
15.15-16.30 X10_ Useful bits and pieces

* All times are EDT
Quo vadis processor architecture?

OUTLINE
Credits

Slides 6 – 26 with the kind permission of Prof. Wolfgang E. Nagel
Quo vadis processor architecture?

- Moore’s law: increasing transistor budget
- challenge: translate transistors into performance
- multiple approaches
  - improve sequential performance
    - increase instruction window of out-of-order execution
    - better branch prediction
    - larger caches
  - increase parallelism
    - wider SIMD instructions
    - multicore processors
Increasing sequential performance is costly

- additional logic and buffers for Out-of-Order Execution
- large tables for branch prediction
- caches consume large chip area in CPUs
- increasing these structures not efficient
  - only moderate ILP improvements in past year

- hardware development focuses on increasing parallelism
  - software development has to follow
More parallelism for more performance

- until 2005 performance improvement primarily driven by frequency
- since then, focus is on increasing parallelism

<table>
<thead>
<tr>
<th>processor</th>
<th>year</th>
<th>frequency [MHz]</th>
<th>ops/cycle</th>
<th>cores</th>
<th>Gflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium Pro (P6)</td>
<td>1995</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>0,2</td>
</tr>
<tr>
<td>Pentium 2 (Klamath)</td>
<td>1997</td>
<td>300</td>
<td>1</td>
<td>1</td>
<td>0,3</td>
</tr>
<tr>
<td>Pentium 3 (Katmai)</td>
<td>1999</td>
<td>600</td>
<td>1</td>
<td>1</td>
<td>0,6</td>
</tr>
<tr>
<td>Pentium 4 (Willamette)</td>
<td>2000</td>
<td>1500</td>
<td>2</td>
<td>1</td>
<td>3,0</td>
</tr>
<tr>
<td>Pentium 4 (Northwood)</td>
<td>2002</td>
<td>3067</td>
<td>2</td>
<td>1</td>
<td>6,1</td>
</tr>
<tr>
<td>Pentium D (Smithfield)</td>
<td>2005</td>
<td>3200</td>
<td>2</td>
<td>2</td>
<td>12,8</td>
</tr>
<tr>
<td>Core2 Duo (Conroe)</td>
<td>2006</td>
<td>2667</td>
<td>4</td>
<td>2</td>
<td>21,3</td>
</tr>
<tr>
<td>1st gen Core i7 (Nehalem)</td>
<td>2008</td>
<td>3200</td>
<td>4</td>
<td>4</td>
<td>51,2</td>
</tr>
<tr>
<td>2nd gen Core i7 (Sandy Br.)</td>
<td>2011</td>
<td>3400</td>
<td>8</td>
<td>4</td>
<td>108,8</td>
</tr>
<tr>
<td>4th gen Core i7 (Haswell)</td>
<td>2013</td>
<td>3400</td>
<td>16</td>
<td>4</td>
<td>217,6</td>
</tr>
</tbody>
</table>
One instruction on multiple data with SIMD extensions

- A single instruction stream controls multiple functional units that handle one data stream each.
Explicit parallelism

- manual programming vs. automatic vectorization
- new implementation/compilation for every extension of vector length
The number of SIMD registers increases.
Use SIMD processor capabilities
Using SIMD instructions - via compiler

- compilers automatically try to vectorize
  - requires compiler flags to specify target ISA (e.g., -xAVX)
  - can significantly improve performance
- block vectorization
  - combines consecutive instructions into a single SIMD instruction
Using SIMD instructions - via compiler

- loop vectorization
  - replaces multiple loop iterations with one using SIMD
  - only works for simple loop constructs (e.g. for(i=0;i<n;i++){…})
- does not always work as expected
  - check compiler output for performance critical loops
  - hard to convince the compiler to do the right thing
    - i.e., with flags, pragmas, etc.
## SIMD performance – according to spec sheet

<table>
<thead>
<tr>
<th>Processor</th>
<th>Xeon X5670</th>
<th>Xeon E3-1280</th>
<th>Core i7 4770</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Westmere</td>
<td>Sandy Bridge</td>
<td>Haswell</td>
</tr>
<tr>
<td>Vector ISA</td>
<td>SSE4.2</td>
<td>AVX</td>
<td>AVX2</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.93 GHz</td>
<td>3.5 GHz</td>
<td>3.4 GHz</td>
</tr>
<tr>
<td>FPU</td>
<td>2x 128 Bit</td>
<td>2x 256 Bit</td>
<td>2x 256 Bit + FMA</td>
</tr>
<tr>
<td>Operations/cycle</td>
<td>4 dp (2 mul, 2 add)</td>
<td>8 dp (4 mul, 4 add)</td>
<td>16 dp (8 fma)</td>
</tr>
<tr>
<td>L1 / L2 per core</td>
<td>2x 32KiB / 256 KiB</td>
<td>2x 32KiB / 256 KiB</td>
<td>2x 32 KiB / 256 KiB</td>
</tr>
<tr>
<td>L3 per chip</td>
<td>12 MiB</td>
<td>8 MiB</td>
<td>8 MiB</td>
</tr>
<tr>
<td>Memory</td>
<td>3x DDR3 1333</td>
<td>2x DDR3 1333</td>
<td>2x DDR3 1600</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel 11.1</td>
<td>Intel 12.0</td>
<td>Intel 14.0</td>
</tr>
<tr>
<td>GFLOPS per core</td>
<td>11,7</td>
<td>28</td>
<td>54,4</td>
</tr>
</tbody>
</table>
Compilers do not exploit max SIMD performance
Manually tuned code comes close to SIMD peak
Explicit SIMD programming is sophisticated

- intrinsics
  - can be used in C
  - `#include <ia32intrin.h>` (meta header)
  - `_mm[256]_<operation>_<datatype>(operands)`
  - additional vector data types that match SIMD registers
Explicit SIMD programming is sophisticated

- inline assembler:
  - more flexible, usage of __asm is sophisticated
  - loops, if-then-else, etc. also need conversion to assembler
  - programmer responsible for register (de-)allocation
  - different implementations for x86 and x86_64 needed
    - due to different register counts and sizes
Multicore processors share auxiliaries

- small IPC/IPS improvements per core
  - increased Out-of-Order window
  - more load/store buffers
- large SIMD improvements per core
  - more ops per instruction
  - wider data paths
- performance driven by core count
- support resources shared by all cores
FP performance grows faster than bandwidth to main memory

- 2002 – 2012
- 12x memory bandwidth
- 38x floating point performance
The cache concept is transparent to apps

- no explicit addressing
- improves application performance without modification
Common properties of caches

• multiple levels
  – increasing capacity
  – larger caches have higher latency

• separate instruction and data caches on level 1
  – required for efficient pipelining
    • inst. fetches do not interfere with load/stores
  – allows lower latency
    • instruction cache near decoder
    • data cache near execution units

• unified lower level caches (level 2-3)
Latencies of random memory access

- L1: 3 cycles
- L2: 15 cycles
- L3: 44 cycles
- RAM: 208 cycles
Bandwidth & latency in memory hierarchy

- L1: 75 GB/s, 3 cycles
- L2: 20 GB/s, 15 cycles
- L3: 10 GB/s, 44 cycles
- RAM: 5 GB/s, 208 cycles

consider cache usage in implementation
Throughput of arithmetic operations

- 2x Xeon E5 2.7 GHz: 345 GFLOPS peak performance
- Benchmarks load one operand from memory and perform:
  - and/add/mul: one floating point operation
  - madd: two floating point operations
- Two operations per memory operand required to achieve peak performance even if data is in L1
Core 2 core vs. L3 Cache latency

- L1 and L2 are per core, L3 is shared
- exchange between cores slower than access to L3 cache
- frequent exchanges limits achievable performance
Bandwidth to L1 and L2 scales linearly

- L1 and L2 replicated per core
- bandwidth scales linearly with core count
L3 cache bandwidth increases with core #

- L3 cache does not scale linearly
- yet, bandwidth increases significantly when using more cores in parallel
- bandwidth is almost saturated with 2 cores
  ⇒ shared resources can limit the scalability
Efficient memory access
Memory properties to keep in mind

- memory forms a hierarchy (L1-L3 cache, main)
- cache closer to the processor is faster but smaller
- data are transferred from main memory to cache, as needed
- cache miss: data retrieved from higher levels of memory
  - expensive (latency)
  - blocks of data occupy a cache line (~64 byte)
  - cached data removed to make space for new data
Memory hierarchy is not controlled by user/compiler

- CPU assumes that memory is accesses sequentially
- time locality
  - organize data accesses so that values are used multiple times
- spatial locality
  - when a value is fetched, nearby elements will also be fetched
Memory is organized sequentially

- In C, a 2-D array is stored in rows (columns in Fortran)
  - Element [0][0] is followed by [0][1], and then by [0][2]
  - When an element is transferred to cache, succeeding row elements (in C) are also transferred (pre-fetching)
  - Matrix-based computation should access the elements of the arrays by rows (and by columns in Fortran)
The ominous Translation-Lookaside Buffer (TLB)

- virtual memory: logical addresses arranged into virtual pages.
- a typical page size is 4 or 8 KB
- physical pages are spread out in memory
  - virtual pages must be mapped to the physical ones.
  - OS maintains a page table, that records this mapping
  - CPU has special cache (TLB) for this table
  - cache miss is called TLB miss
Loop Optimization

OUTLINE
A different order of nested loops can be faster

- i-loop with embedded j-loop may be faster if loops are switched
  - C: data is accessed in rows instead of in columns.
- most computational time is spent in loops on arrays
  - consider reorganization to exploit cache
- do not change loop order if the loop nest has
  - memory locations referenced more than once
  - at least one of those references modify its value
A different statement order might make a difference

- check if a reordering of statements is allowed
  - sometimes better by programmers than compilers
  - consider reordering if memory accesses to array elements does not reflect their order in memory
  - consider reordering if loop has a large body and references to an array element or its neighbors are far apart.
  - better utilization of the instruction pipeline
  - might increase the size of a parallel area (SIMD)
Loop unrolling enlarges the loop body

- packs work of several loop iterations into a single pass
  - reduces the overheads of loops
  - reduce the number of passes through the loop by a factor
    - “unroll factor”
    - reduced by unroll factor:
      - increment of the loop variable
      - test for completion
      - branches to the start of the loop code
  - good for
    - cache line utilization
    - vectorization
    - instruction level parallelism (ILP)
Let the compiler do the unrolling job

- compilers are good at doing loop unrolling
  - set respective flags and check the compiler report
- manual unrolling kills portability and maintainability of your code
- loop already contains a lot of computation:
  - unrolling may render the cache less efficient
- loop contains procedure calls:
  - unrolling might result in new overheads
- loop contains branches:
  - the benefits may be low
Loop fusion merges loops to create a bigger loop

- may enable data in cache to be reused more frequently
- might increase the amount of computation per iteration
  - improve instruction level parallelism
- reduces loop overhead
  - more work is done per loop
Loop fission breaks up a loop into several loops

- might improve locality
- may improve the use of cache
- may isolate a part that inhibits the full optimization of the loop
- most useful when loop nest is large and does not fit into cache
  - instruction cache
  - data cache
- optimize different loop parts in different ways
Loop blocking/tiling ensures that data stays in cache

- goal: improve temporal and spatial locality, i.e. data access/reuse
- re-organizes data accesses into chunks/tiles
- indication:
  - large data sizes and bad memory access
  - little data reuse in the loop
- replaces the original loop with a couple of loops
- size of the tiles highly depends on the cache sizes
Points and contiguous memory in C

- Pointers are used in C as arrays.
- They pose a serious problem for compilers.
- The compiler assumes that pointers can access any memory location.
  - Pointer aliasing problem.
  - Prevents the compiler from optimizing.
    - Cannot determine safety of optimization.
- A guarantee that a pointer points to non-overlapping memory helps.
  - Distinct calls to `malloc()` function.
  - Compiler directives/options.
The bottom line: make the compiler do the job

- most tuning techniques are implemented in modern compilers
- compiler decides what technique to use
- experiment with compiler options (on critical code)
- options differ a great deal between compilers
- compiler is limited by its ability to analyze the program’s intentions
  - provide hints
  - keep code simple and easy (for the compiler)
- compiler cannot improve memory usage of nonlocal data structures
  - rearranging of source code might help
QUESTIONS?

https://connect.iu.edu/ptune15
Class evaluation

- we appreciate your opinion and feedback
- please consider filling-in our class evaluation questionnaire

https://www.surveymonkey.com/r/ptune15