Introduction to Performance Tools

By Brent Swartz

swartzbr@umn.edu
Outline

• Definitions
  • Performance
    • Bandwidth
    • FLOPS
    • GHz
    • Cores (multi-core processor)
    • Theoretical Peak vs Achieved
    • AVX (vectors)
    • Time to Science (TTS)
• Roofline Model
• Optimizations
  • Cache Blocking
  • Amdahl’s Law
• Scaling
  • Weak vs Strong
  • Problem size vs proc count
  • Load Balancing
  • Know when to stop scaling
Outline

• Compiler Options
  • Importance of Code Verification/Test
• Code Optimizations
  • Identify kernels in code (e.g., nested loops for mat-mul)
  • Leverage libraries like BLAS/LAPACK, MKL
  • Vectorization
• Improving Code Scalability
  • Load balancing
  • Leverage domain decomposition tools like ParMETIS
• Basic Profiling: Obtain baseline
  • 90-10 Split
• Generic Performance Tools
  • gprof
• AMD Performance Tools
  • AMD uProf: https://developer.amd.com/amd-uprof/
Outline

• Intel Performance Tools
  • Advisor: (Overview at: https://software.intel.com/en-us/intel-advisor-xe)
    • https://software.intel.com/en-us/advisor-user-guide-workflows
  • Vectorization Optimization:
    https://software.intel.com/en-us/advisor/features/vectorization
    • Memory Access Pattern Analysis (MAP)
      https://software.intel.com/en-us/advisor-tutorial-vectorization-window-
      s-cplusplus-analyze-memory-access-patterns
  • Thread Prototyping:
    https://software.intel.com/en-us/advisor/features/threading
  • Roofline model:
  • Flow Graph Analyzer (For TBB):
Outline

- Intel Performance Tools
  - VTune Amplifier (Overview at: https://software.intel.com/en-us/vtune )
    Start with:
    - Performance Snapshot:
  Others:
    - Diagnose memory, NUMA, and I/O:
      https://software.intel.com/en-us/vtune/features/memory-storage
- MPI Trace Analyzer and Collector
Definitions

Performance

- Bandwidth: GB/s is GigaBytes per second.
- GHz: GigaHertz, Billions of cycles per second.
- Multi-core processor: A **multi-core processor** is a single **computing** component with two or more independent **processing units** called cores, which read and execute **program instructions**. The instructions are ordinary **CPU instructions** (such as add, move data, and branch) but the single processor can run multiple instructions on separate cores at the same time, increasing overall speed for programs amenable to **parallel computing**. Manufacturers typically integrate the cores onto a single **integrated circuit die** (known as a chip multiprocessor or CMP) or onto multiple dies in a single **chip package**. An example of a multi-core processor is one socket of the two per node on Mesabi, containing 12 cores, and sharing a 30 MB L3 cache.

Source: [https://en.wikipedia.org/wiki/Multi-core_processor](https://en.wikipedia.org/wiki/Multi-core_processor)
Definitions

Nehalem core.

Source: http://www.setcom.ee/tanno/info/pictures/409.html
Definitions

Source: https://en.wikichip.org/wiki/File:haswell_block_diagram.svg
Definitions

Performance

• Theoretical Peak vs Achieved:
  Theoretical Peak is how many floating-point operations the system can theoretically execute in a given time. Recent Intel CPUs (starting with Haswell) have a different base frequency when exploiting certain instructions, i.e. AVX instructions.

  Using the maximum AVX frequency at 24 cores, the Mesabi node theoretical peak is: 24 cores * 16 FLOP/cycle * 2.8 GHz = 1075.2 GFLOPS

• Vectorization is the operation of Single Instruction Multiple Data (SIMD) instructions (like Intel® Advanced Vector Extensions) on multiple data objects in parallel within a single CPU core. This can greatly increase performance by reducing loop overhead and making better use of the multiple math units in each core. The effectiveness of vectorization depends on many factors including data alignment, loop iteration count, the complexity of the control flow instructions, etc.

Source: www.dolbeau.name/dolbeau/publications/peak.pdf
Definitions

Performance

- AVX/AVX2: Intel Advanced Vector eXtensions. The third SIMD (vector) instruction set for the x86 architecture. It uses 256 bit wide registers. The original AVX instruction set only support floating point operations. AVX2 introduced integer operations and 3 operand instructions, such as FMA (Floating-point Multiply Add).

- Time to Science (TTS): The time it takes to do the computing required to advance your science, i.e. the time to get results.

Definitions

• Roofline Model

The Roofline model is an intuitive visual performance model used to provide performance estimates of a given compute kernel or application running on multi-core, many-core, or accelerator processor architectures, by showing inherent hardware limitations, and potential benefit and priority of optimizations. By combining locality, bandwidth, and different parallelization paradigms into a single performance figure, the model can be an effective alternative to assess the quality of attained performance instead of using simple percent-of-peak estimates, as it provides insights on both the implementation and inherent performance limitations.

- Arithmetic Intensity (AI) = Flops / Bytes (as presented to DRAM)
- A given kernel/application will have an AI (or Operational Intensity) determined by how many Flops it does per byte of memory transferred from DRAM.

http://dl.acm.org/citation.cfm?doid=1498765.1498785
Definitions


By Giu.natale - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=49641314
Definitions

- Optimizations
  - Amdahl’s Law: **Amdahl's law** is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law can be formulated in the following way:
    \[ SP( s ) = \frac{1}{(1 - p) + \frac{p}{s}} \]
    Where
    - \( SP \) is the theoretical speedup of the execution of the whole task;
    - \( s \) is the speedup of the part of the task that benefits from improved system resources;
    - \( p \) is the proportion of execution time that the part benefiting from improved resources originally occupied.

Fundamentally Amdahl’s Law says that the speedup of an application running on many cores is limited to \( \frac{1}{(\text{percent serial code in the application})} \), e.g. an app with 10% serial code cannot go more than 10 times faster, no matter how many cores it runs on. So the serial part of the code should be minimized.

Source: https://en.wikipedia.org/wiki/Amdahl%27s_law
Amdahl’s Law

Source: https://en.wikipedia.org/wiki/Amdahl%27s_law
Definitions

- Optimizations
  - Cache Blocking:
    
    An important class of algorithmic changes involves blocking data structures to fit in cache. By organizing data memory accesses, one can load the cache with a small subset of a much larger data set. The idea is then to work on this block of data in cache. By using/reusing this data in cache we reduce the need to go to memory (reduce memory bandwidth pressure).

Definitions

• Scaling
  • Weak vs Strong:
    In the context of high performance computing there are two common notions of scalability:
    - The first is strong scaling, which is defined as how the solution time varies with the number of processors for a fixed total problem size.
    - The second is weak scaling, which is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

Source: https://en.wikipedia.org/wiki/Scalability#Weak_versus_strong_scaling
Definitions

**Gustafson's law** can be formulated the following way:

\[ S \text{ latency } (s) = 1 - p + s \cdot p, \]

where

- \( S \text{ latency} \) is the theoretical speedup in latency of the execution of the whole task;
- \( s \) is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- \( p \) is the percentage of the execution workload of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

The impact of Gustafson's law was to shift research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In a way the law redefines efficiency, due to the possibility that limitations imposed by the sequential part of a program may be countered by increasing the total amount of computation.

- [https://en.wikipedia.org/wiki/Gustafson%27s_law](https://en.wikipedia.org/wiki/Gustafson%27s_law)
**Definitions**

**Gustafson's law** can be formulated the following way:

\[ S \text{ latency (s)} = 1 - p + s \times p, \]

where

- \( S \text{ latency} \) is the theoretical speedup in latency of the execution of the whole task;
- \( s \) is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- \( p \) is the percentage of the execution \textit{workload} of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

Gustafson's law addresses the shortcomings of **Amdahl's law**, which is based on the assumption of a fixed \textit{problem size}, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

The impact of Gustafson's law was to shift [citation needed] research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In a way the law redefines efficiency, due to the possibility that limitations imposed by the sequential part of a program may be countered by increasing the total amount of computation.

- [https://en.wikipedia.org/wiki/Gustafson%27s_law](https://en.wikipedia.org/wiki/Gustafson%27s_law)
Definitions

• Load Balancing: **Load balancing** improves the distribution of **workloads** across multiple computing resources, such as computers, a **computer cluster**, **network links**, **central processing units**, or **disk drives**.\(^1\) Load balancing aims to optimize resource use, maximize **throughput**, minimize response time, and avoid overload of any single resource.

• Load balancing an application for HPC usually involves:
  
  • (For OpenMP) Ensuring each thread in a work-sharing construct has an equal amount of work.
  
  • (For MPI) Ensuring each MPI task in an MPI program has an equal amount of work.

  In both cases, the goal is for each thread/task to complete at the same time, so that individual threads/tasks are not left working while the others are idle.

Source: [https://en.wikipedia.org/wiki/Load_balancing_(computing)](https://en.wikipedia.org/wiki/Load_balancing_(computing))
Compiler Options

- Compiler Options (Intel)

From the Intel documentation:

-ax Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel processors if there is a performance benefit.

-x Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

-march Tells the compiler to generate code for processors that support certain features.

-mtune Performs optimizations for specific processors.

The usual way is to use either one of the first two (-ax or -x), or the last two as a pair. The easiest solution is probably the special architecture parameter “native” to -march & -mtune 3, which instructs the compiler to generate and optimize code for whatever features is supported by the current CPU. As long as a code is compiled on the machine it will run on, it will always use all available features. If cross compiling, then a specific kind of architecture should be picked, i.e. for AVX2 -march=core-avx2 -mtune=core-avx2.
Compiler Options

• Compiler Options (Intel)

Also see: https://software.intel.com/en-us/forums/intel-c-compiler/topic/731338

Since MSI has Intel Haswell Xeon processors on Mesabi, and will have AMD EPYC ROME processors on the new Mangi cluster, you can compile for both using these options:

-axCORE-AVX2,SSE4.2

Note: A complete description of all possible compiler options (Intel or GNU) is beyond the scope of this tutorial.
Compiler Options

- Compiler Options (GNU)

GNU compiler: Recent GNU compilers “gcc” offer a similar interface to the Intel compiler for -march & -mtune, and also support the special “native” architecture. It does not have the -ax or -x options, but it offers a large set of instruction set-specific switches, such as -mavx, -mavx2 or -maes. e.g. gcc -march=haswell

Note that you can use the option -mdump-tune-features.

This option instructs GCC to dump the names of the x86 performance tuning features and default settings. The names can be used in -mtune-ctrl=feature-list.

-mtune-ctrl=feature-list

This option is used to do fine grain control of x86 code generation features. feature-list is a comma separated list of feature names.

For more info see:

https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/
https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html#Invoking-GCC
http://gcc.gnu.org/onlinedocs/gcc/x86-Options.html
Code Verification

- Code Verification/Test

   It is very important to do verification/testing of your code after code modification and/or compiler option modification. The recommended approach is to:

   • Create an input test case that exercises the relevant code that you are modifying. It helps to size the input so that running the code doesn’t take too long or short of a time (say, 10 to 60 minutes).
   • Execute the code on the relevant cluster to get a baseline timing/profile and reference output.
   • Make code/compiler option modifications. Focus code modifications on the functions/routines which consume the most time.
   • Test the code by executing it, and verifying the output is the same, and generating a new timing/profile.

Through this iterative process, you can ensure that the program output is still correct, while determining the effect of modifications on the timing/profile. It inherently limits the amount of code that has been modified since the last known good run, so that any errors introduced are more easily tracked down.

For more info, see:  [https://en.wikipedia.org/wiki/Software_verification_and_validation](https://en.wikipedia.org/wiki/Software_verification_and_validation)
Here is an outline of how to optimize your code:

1. Identify kernels in code (e.g., nested loops for mat-mul) using gprof (described later) or other (performance snapshot) profiling tools:
   
   

2. If possible, leverage libraries like BLAS/LAPACK, FFTW, etc. from the relevant vendor. In general these routines have been heavily optimized for the vendor’s platform (so you won’t have to!):
   
   
3. Vectorization (cont.):

Use compiler options to help see whether loops are vectorized (and why/why not):

- For Intel compilers: To get information on whether a loop was vectorized or not, enable generation of the optimization report using the options:
  - qopt-report=1
  - qopt-report-phase=vec

  These options generate a separate report in an *.optrpt file that includes optimization messages. Alternatively, you can use:
  - ifort -guide-vec=4 code.f

- For GNU compilers: Use `-Wvector-operation-performance`
  - Using `-O3` vs `-O2` can enable other vectorization options (so both `-O2` and `-O3` should be tested).
  - Can also try: `-fvect-cost-model=cheap`

Can use Vectorization Advisor (in the Intel Advisor tool):

Code Optimization

• Improving Code Scalability
  • Load balancing
    • For MPI task load balancing, can use:
    • For thread load balancing, see:
    • Can check for thread load imbalance with:
      https://software.intel.com/en-us/advisor/features/threading

• Leverage domain decomposition tools.
  There exist domain decomposition tools like ParMETIS (http://glaros.dtc.umn.edu/gkhome/node/96) which can help achieve load balance. Others can be found in:
Generic Performance Tools

• Gprof
  • How to Use gprof:
    1. Compile and link with the -pg option.
    2. Run your program normally; that is, use representative input (and check difficult, slow, or fast cases).
    3. Type gprof exec > out where exec is replaced by your executable's name and out by some meaningful name for the profiling information. For instance, if your executable were "foo" on the third run compiled with the -O2 option then you might type:

        gprof foo > run3.withO2.stats

    4. Look over the output and learn what it means. Hint, go to the second table. The first is pretty useless. To get there, search for "granularity" twice from the top of the file (in emacs, use C-s; in less or more use /).

Source: https://users.cs.duke.edu/~ola/courses/programming/gprof.html
Generic Performance Tools

- Gprof Notes:
  1. Leave out the -g option and maybe even include the -O (or -O2 or -O3) option in your compilation. Otherwise, you can get biased timing information.
  2. Optimize the big time-suckers and the high frequency called functions first! That's where you make your big gains.
  3. Type “gprof exec > out” where exec is replaced by your executable's name and out by some meaningful name for the profiling information. For instance, if your executable were "foo" on the third run compiled with the -O2 option then you might type: gprof foo > run3.withO2.stats
Code Optimization

- AMD Performance Tools
  - AMD uProf:  https://developer.amd.com/amd-uprof/
Improving Code Scalability

- Technique to help codes use more nodes efficiently (scale up)
  - Uses a profiler like gprof to get timings for individual routines
1. Profile the code using X nodes.
2. Profile the code (with the same input) using 2 * X nodes.
3. Create a table showing the sorted RATIO of the routine times from the 2X profile to the 1X profile.

The routines at the top of the list (the highest ratios) are the routines (in order) that are not scaling well. Focus on these routines as candidates for improvement. Causes can be MPI routines (use an Infiniband aware MPI), I/O, load imbalance, or other causes.
Summary

- See the links in the Outline for the Intel Performance Tools info.

For support at MSI, contact help@msi.umn.edu