INTRODUCTION TO PERFORMANCE TUNING AND OPTIMIZATION TOOLS

Bei Wang
beiwang@princeton.edu
Princeton University

Third Computational and Data Science School for HEP (CoDaS-HEP 2019)
July 25, 2019
Outlines

• **Basic Concepts in Performance Tuning**
  • What is performance tuning and why it matters?
  • Performance tuning workflow
  • Typical pitfalls wrt. single node performance
  • Performance tool overview

• **Performance Tools: Demos and Hands-on**
  • How to run basic timing experiments and what they can do
  • How to use hardware counters
  • How to deal with parallelism (vectorization and threads)

• **Goals**
  • Provide basic guidance on how to understand the performance of a code using tools
  • Provide starting point for performance optimizations
BASIC CONCEPTS IN PERFORMANCE TUNING
What Is and Why Performance Tuning?

• What is performance tuning?
  • The process of improving the efficiency of an application to better utilize a given hardware resource
  • Requires some understanding about the performance features of the given hardware (see CoDaS’s talk “what every physicist should know about computer architecture” on Monday)
  • Identifying bottlenecks, determining efficiency and eliminating the bottlenecks if possible
  • Incrementally complete tuning until the performance requirements are satisfies

• Why performance matters?
  • Energy efficiency
  • Today’s applications only use a fraction of the machine due to
    • Complex architectures
    • Mapping applications onto architectures is hard
Performance Tuning Workflow

- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all **changes**
- Apply regression test to **ensure correctness** after each change
- Remember: fast computing of wrong result is completely irrelevant

- Choose an workload which is measurable, representative, static and reproducible, and quantifiable
- Record code generation, compiler version, compiler flags, input parameters, core count, affinity etc

**prepare**

**modify**

**measure**

**analyze**

**hypothesize**
Measure

• What to measure? Choose metrics which quantify the performance of your code
  • Time, energy etc

• How to measure?
  • Linux “time” command
    • Get an idea of overall run time, but can’t pin performance bottlenecks
  • Put timer (e.g., gettimeofday, MPI_Wtime, omp_get_wtime) around loops/functions
    • Works for small code base to identify hotspots, but hard to maintain and require significant priori knowledge
  • Performance tools (recommended)
    • Collect a lot data with varying granularity, cost and accuracy
    • Trace back to source code (use –g compiler flag)
  • How to collect

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Records system state at periodic intervals</td>
<td></td>
</tr>
<tr>
<td>• Useful to get an overview</td>
<td></td>
</tr>
<tr>
<td>• Low and uniform overhead</td>
<td></td>
</tr>
<tr>
<td>• Ex. Profiling</td>
<td>• Records all events</td>
</tr>
<tr>
<td></td>
<td>• Provide detailed per event information</td>
</tr>
<tr>
<td></td>
<td>• High overhead for request events</td>
</tr>
<tr>
<td></td>
<td>• Ex. Tracing</td>
</tr>
</tbody>
</table>

• Sometime there is a learning curve to master the tools
Performance Tools Overview

- **Basic OS tools**
  - Time
  - Gprof/perf
  - Valgrind/callgrind

- **Hardware counter**
  - PAPI API & tool set

- **Community open source**
  - HPCToolkit (Rice Univ.)
  - TAU (U of Oregon)
  - Open|SpeedShop (Krell)

- **Commercial products**
  - ARM MAP
  - Intel VTune Amplifier
  - Intel Advisor
  - Intel Trace Analyzer

- **Vendor supplied (free)**
  - CrayPat
  - Nvprof/pgprof

No tool can do everything. Choose the right tool for the right task.
Typical Pitfalls wrt. Performance: Sequential

- Where am I spending my time?
  - Find the hotspots

- Is my code computational or memory bounded?
  - Memory bounded
    - Data locality
    - TLB misses
    - L1/L2/L3 $ misses
  - Computational bounded
    - Fast math (see CoDaS’s talk “Floating Point Arithmetic” on Wed)
    - Avoid type conversion
      - float x=3.14;  // bad: 3.14 is a double
      - float s=sin(x);  // bad: sin() is a double precision function
      - long v=round(x);  // bad: round takes and returns double
      - float x=3.14f;  // good: 31.4f is a float
      - float s=sinf(x);  // good: sin() is a single precision function
      - long v=lroundf(x);  // good: lroundf() takes float and returns long
  - Vectorization efficiency

- Is my I/O efficient?
Typical Pitfalls wrt. Performance: Multithreading

- Load imbalance
- False sharing
  - Occurs when threads on different processors modify variables that reside on the same cache line
  - Caused by coherent caches
  - Cache line is 64 bytes wide
- Insufficient parallelism
- Synchronization
  - Avoid synchronization with private thread storage
- Non-optimal memory placement
  - Thread affinity
  - Allocation on first touch

LINUX TOOL: Perf
Perf is a performance analyzing tool in Linux, available in version 2.6.31

How does it work

- **perf record**: measure and save sampling data for a single program
  - `-g`: enable call-graph (callers/callee information)
- **perf report**: analyze the file generated by perf record, can be flat profile or graph
  - `-g`: enable call-graph (callers/callee information)
- **perf list**: list available events for measurement
  - Support a list of hardware and software events
- **perf stat**: measure total event count for a single program
  - `-e`: event names provided in perf list

etc

When compiling the code, use the following flags for easier interpretation

- `-g`: need debug symbols in order to annotation source
- `-fno-omit-frame-pointer`: provide stack chain/backtrace
Example: Matrix-Matrix Multiplication

Two versions of 2D matrix-matrix multiplication

```c
int main(int argc, char *argv[]) {
    int matrix_size; //N*N matrix
    int max_iters=10; //number of times to call a matrix-matrix function

    //read command line input
    //set various parameters
    if(argc<2) {
        cout<<"ERROR: expecting integer matrix size, i.e., N for NxN matrix"<<endl;
        exit(1);
    } else {
        matrix_size=atoi(argv[1]);
    }

cout<<"using matrix size:"
matrix_size<<endl;

double **A, **B, **C; //2D arrays
create_matrix_2D(A, B, C, matrix_size);
init_matrix_2D(A, B, C, matrix_size);
for (int r=0; r < max_iters1 r++) {
    zero_result(C, matrix_size);
    #ifdef NAIVE
        compute_naive(A,B,C,matrix_size);
    #elif INTERCHANGE
        compute_interchange(A,B,C,matrix_size);
    #endif
    free_matrix_2D(A, B, C, matrix_size);
    return 0;
}
```
Set up Adroit for Hands-on

- How to log into the Adroit system
  - Login information was distributed on Monday
- Download the exercises from Github
  - `git clone https://github.com/beiwang2003/codas_perftools.git`
- Move to the `codas_perftools` directory
  - `cd $HOME/codas_perftools`
- Load environment module
  - `module load rh/devtoolset/7`
Hands-on: Find Hot Spots Using Perf

- Compile the code: `g++ -g -fno-omit-frame-pointer -O3 -DDNAIVE matmul_2D.cpp -o mm_naive.out`
- Collect profiling data: `perf record -g ./mm_naive.out 500`
- Open the result: `perf report -g`
Hands-on: Loop Interchange Optimization

- The `perf list` command lists all available CPU counters:

<table>
<thead>
<tr>
<th>Event Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch-instructions OR branches</td>
<td>Hardware event</td>
</tr>
<tr>
<td>branch-misses</td>
<td>Hardware event</td>
</tr>
<tr>
<td>bus-cycles</td>
<td>Hardware event</td>
</tr>
<tr>
<td>cache-misses</td>
<td>Hardware event</td>
</tr>
<tr>
<td>cache-references</td>
<td>Hardware event</td>
</tr>
<tr>
<td>cpu-cycles OR cycles</td>
<td>Hardware event</td>
</tr>
<tr>
<td>instructions</td>
<td>Hardware event</td>
</tr>
<tr>
<td>ref-cycles</td>
<td>Hardware event</td>
</tr>
<tr>
<td>alignment-faults</td>
<td>Software event</td>
</tr>
<tr>
<td>bpf-output</td>
<td>Software event</td>
</tr>
<tr>
<td>context-switches OR cs</td>
<td>Software event</td>
</tr>
<tr>
<td>cpu-clock</td>
<td>Software event</td>
</tr>
<tr>
<td>cpu-migrations OR migrations</td>
<td>Software event</td>
</tr>
<tr>
<td>dummy</td>
<td>Software event</td>
</tr>
<tr>
<td>emulation-faults</td>
<td>Software event</td>
</tr>
<tr>
<td>major-faults</td>
<td>Software event</td>
</tr>
<tr>
<td>minor-faults</td>
<td>Software event</td>
</tr>
<tr>
<td>page-faults OR faults</td>
<td>Software event</td>
</tr>
<tr>
<td>task-clock</td>
<td>Software event</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>L1-dcache-stores</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>LLC-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>LLC-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>LLC-stores</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>branch-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>branch-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>dTLB-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>dTLB-store-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>dTLB-stores</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>sTLB-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>sTLB-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>sTLB-stores</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>node-load-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>node-loads</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>node-store-misses</td>
<td>Hardware cache event</td>
</tr>
<tr>
<td>node-stores</td>
<td>Hardware cache event</td>
</tr>
</tbody>
</table>

- Check `man perf_event_open` to see what does each event measure

- The `perf stat` command instruments and summarizes selected CPU counters

1. Compile the code
   - `g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out`

2. Run `perf stat`
   - `perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores ./mm_naive.out 500`

3. Record the numbers for each events
4. Compile the code
   - `g++ -g -fno-omit-frame-pointer -O3 -DINTERCHANGE matmul_2D.cpp -o mm_interchange.out`

5. Run `perf stat`
   - `perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses ./mm_interchange.out 500`

6. Compare the numbers for both cases
## Results Comparison (GCC)

### NAIVE

Performance counter stats for `./mm_naive.out 500`:

<table>
<thead>
<tr>
<th>CPU Cycles</th>
<th>Instructions</th>
<th>L1-dcache-loads</th>
<th>L1-dcache-load-misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,564,503,540</td>
<td>10,063,662,841</td>
<td>3,767,490,743</td>
<td>1,475,374,174</td>
</tr>
</tbody>
</table>

1.691104619 seconds time elapsed

### INTERCHANGE

Performance counter stats for `./mm_interchange.out 500`:

<table>
<thead>
<tr>
<th>CPU Cycles</th>
<th>Instructions</th>
<th>L1-dcache-loads</th>
<th>L1-dcache-load-misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,589,454,237</td>
<td>8,869,823,983</td>
<td>3,143,807,817</td>
<td>164,669,312</td>
</tr>
</tbody>
</table>

0.787522929 seconds time elapsed

- The number of CPU cycles is much lower for interchange, reflecting its shorter elapsed time
- The number of instructions are half in interchange
- Interchange has substantially fewer LL1 load misses, which indicates better data locality

Follow up exercise: change matrix dimension to 1000x1000. This will trigger more LLC and TLB misses.
OPEN|SpeedShop
OpenSpeedShop (O|SS)

- Open source multi-platform performance tool
  - Available on Intel, AMD, ARM, Power PC, Power 8, GPU based systems
  - Built on top of a list of community tools, e.g., Dyninst and MRNet from UW, libmonitor from Rice, and PAPI from UTK
- O|SS gathers
  - High level summary: *cbtfsummary* “normal app run script”
  - Program counter sampling: *osspcsamp* “…
  - Call path analysis: *ossusertime* “…
  - Hardware performance counters: *osshwcsamp* “…
  - OpenMP profiling and analysis: *ossomptp* “…
  - MPI profiling and tracing: *ossmpi[p][t]* “…
  - I/O profiling and tracing: *ossio[p][t]* “…
  - Memory analysis: *ossmem* “…
- O|SS displays with
  - GUI: openss -f ./*.openss
  - CLI: openss -cli -f ./*.openss
Osspcsamp: Flat Profile Overview

- Start with flat profile overview
- Flat profile: time spent per functions or per statements
- Collect profiling data: `osspcsamp "./mm_naive.out 1000"` (this will generate a *.openss file)
- Open the result in GUI: `openssl -f ./mm_naive.out-pcsamp-0.openss`
Ossusertime: Call Graph Analysis

- Flat profile does not help you:
  - Distinguish routines called from multiple callers
  - Understand the call invocation history
- Stack traces: caller/callee relationships, inclusive/exclusive time
- Collect profiling data: `ossusertime './mm_naive.out 1000'` (this will generate a *.openss file)
- Open the result in GUI: `openss -f ./mm_naive.out-usertime-0.openss`
Osshwcsamp: Hardware Performance Counters

• Timing information shows where you spend your time. BUT, it doesn’t show you why
• Hardware performance counters: PAPI events (use papi_avail to check available events)
• Collect profiling data: osshwcsamp “./mm_naive.out 1000” PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_L1_DCM (up to 6 events, this will generate a *.openss file)
• Open the result in CLI: openss -cli-f ./mm_naive.out-hwcsamp-0.openss
• View the result with: openss>>expview

```
[beiwang@adroit4 codas_perftools]$ openss -cli "mm_naive.out-hwcsamp-1.openss"
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

<table>
<thead>
<tr>
<th>Exclusive CPU time in seconds</th>
<th>% of CPU Time</th>
<th>papi_tot_cyc</th>
<th>papi_tot_ins</th>
<th>papi_l1_dcm</th>
<th>Comp. Intensity</th>
<th>papi_tot_cyc%</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.480000</td>
<td>99.724518</td>
<td>47291712380</td>
<td>80007419256</td>
<td>13917395159</td>
<td>1.691785</td>
<td>99.804480</td>
<td>compute naive(double**, double**, double**, int) (mm_naive.out: mm.h,20)</td>
</tr>
<tr>
<td>0.010000</td>
<td>0.006871</td>
<td>23214861</td>
<td>59870626</td>
<td>165350</td>
<td>2.578978</td>
<td>0.048993</td>
<td>__random (libc-2.17.so)</td>
</tr>
<tr>
<td>0.010000</td>
<td>0.006871</td>
<td>9831635</td>
<td>19114751</td>
<td>68522</td>
<td>1.944299</td>
<td>0.020749</td>
<td>__random_r (libc-2.17.so)</td>
</tr>
<tr>
<td>0.010000</td>
<td>0.006871</td>
<td>32721784</td>
<td>53848956</td>
<td>9349092</td>
<td>1.645661</td>
<td>0.069056</td>
<td>__memset_sse2 (libc-2.17.so)</td>
</tr>
<tr>
<td>0.010000</td>
<td>0.006871</td>
<td>26877580</td>
<td>64800904</td>
<td>7789482</td>
<td>1.666850</td>
<td>0.056722</td>
<td>__brk (libc-2.17.so)</td>
</tr>
<tr>
<td>14.520000</td>
<td>100.000000</td>
<td>47384358240</td>
<td>80185854939</td>
<td>1393467605</td>
<td>1.692226</td>
<td>100.000000</td>
<td>Report Summary</td>
</tr>
</tbody>
</table>
```

openss>>
Ossomptp: OpenMP Parallel Region

• For parallel execution, is there any load imbalance issue? How do you find the potential cause?
• OMPT API: record task time, idleness, barrier, wait barrier per OpenMP parallel region
• Let’s look at the matrix-matrix example, but now we only compute the result for the upper triangular

```c
//TRIANGULAR: only compute the result for the upper triangular
__attribute__((noinline)) void compute_triangular(double **A, double **B, double **C, int matrix_size) {
  #pragma omp parallel for
  for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size-i; j++) {
      for (int k = 0; k < matrix_size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

• Compile the code: `g++ -g -O3 -fopenmp -DTRIANGULAR matmul_2D.cpp -o mm_triangular_omp.out` (export OMP_NUM_THREADS=4)
• Collect profiling data: `ossomptp "./mm_triangular_omp.out 1000"` (this will generate a *.openss file)
Ossomptp: OpenMP Parallel Region

- Open the result in GUI: `openssl -f ./mm_triangular_omp.out-omptp-0.openss`
Using OMP Clause “schedule(dynamic)”

```c
// TRIANGULAR: only compute the result for the upper triangular
_attribute_((noinline)) void compute_triangular(double **A, double **B, double **C, int matrix_size) {
  #pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size-i; j++) {
      for (int k = 0; k < matrix_size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
#endif
```

WAIT_BARRIER time has reduced significantly (from 5.6s to 0.35s)
Another Important Focus: Efficient Vectorization

• The CoDaS’s talk “Vector Parallelism for Kalman-Filter-Based Particle Tracking on Multi- and Many-Core Processors” has covered many important aspects of vectorization
• This lecture will mainly focus on how to examine vectorization efficient using tools, e.g., Intel Advisor
• Analysis tools:
  • Compiler vectorization report
    • GCC: -fopt-info-vec
    • Intel: -qopt-report=5
  • Look at assembly code
  • Measure performance with PAPI counters, e.g., PAPI_DP_OPS, PAPI_VEC_DP etc
  • Intel Advisor
Intel Advisor
Vectorization Advisor & Roofline

• Vectorization advisor
  • Provide vectorization information from vectorization report
  • Identify the hotspots where your efforts pay off the most
  • Provide call graph information
  • Identify the performance and vectorization issues
  • Check memory access pattern
  • Check dependencies
  • More …

• Roofline
  • How much performance is being left on the table
  • Where are the bottlenecks
  • Which can be improved
  • Which are worth improving
Workflow of Vectorization Advisor

- **Survey**: find the vectorization information for loops and provide suggestions for improvement
- **Trip Counts**: generate a Roofline Chart
- **Memory Access Patterns (MAP)**: see how you access the data
- **Dependencies**: determine if it is safe to force vectorization

Select loops with potential dependencies or inefficient memory access pattern
Survey Analysis

- Compile the code: `icpc -g -O3 -xhost -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out`
- Collect the survey data: `advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000`
- Open the result in GUIL: `advixe-gui mm-advisor`
Dependency Analysis

- Check dependency: `advice-cl -c dependencies -mark-up-list=3 -project-dir ./mm-advisor -- ./mm_interchange_icpc.out 1000`
- Open the result in GUI: `advice-gui mm-advisor`
Resolve Point Aliasing

• We can help the compiler to resolve the dependency complaining caused by point aliasing by:
  • “restrict” keyword and -restrict -std=c90 compiler flag
  • #pragma (GCC) ivdep
  • #pragma omp simd

• We choose OpenMP simd pragma here

```c
//INTERCHANGE: 2D matrix-matrix multiplication
__attribute__((noinline)) void compute_interchange(double **A, double **B, double **C, int matrix_size) {
    for (int i = 0; i < matrix_size; i++) {
        for (int k = 0; k < matrix_size; k++) {
            #pragma omp simd
            for (int j = 0; j < matrix_size; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```
Re-run Survey Analysis

- Compile the code: `icpc -g -O3 -xhost -qopenmp-simd -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out`
- Collect the survey data: `advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000`
- Open the result in GUI: `advixe-gui mm-advisor`
Using 512-bit ZMM register

• Compile the code: `icpc -g -O3 -xhost -qopenmp-simd -qopt-zmm-usage=high -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out`

• Collect the survey data: `advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000`

• Open the result in GUI: `advixe-gui mm-advisor`
Trip Counts Analysis

- Collect the trip counts data
  - `advixe-cl -c tripcounts -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000`
  - Note: we need to first carry out “survey” analysis and use the same project directory for “tripcounts”

- **Trip Counts** analysis shows you loop trip counts and call counts
  - The best vectorization requires the scalar trip count to be divisible by the vector length, or you get remainder loops
  - Call counts amplify the importance of tuning a given loop
• Trip counts analysis also collects **FLOPS** (FLoating-point Operations Per Seconds)
• Collecting FLOPS allows the plotting of a Roofline chart
  
  A visual representation of application performance in relation to hardware limitations, including memory bandwidth and computational peaks
• The horizontal axis is Arithmetic Intensity, a measurement of FLOPs per byte accessed. The vertical axis is performance.
• Provide performance insights
  • Highlights poor performing loops
  • Shows performance “headroom” for each loop
    • Which can be improved
    • Which are worth improving
  • Shows likely causes of bottlenecks
  • Suggest next optimization steps
struct Particle {
    float x, y, z;
    float vx, vy, vz;
};

for (int i = 0; i < nParticles; i++) {
    // Components of the gravity force on particle i
    float Fx = 0, Fy = 0, Fz = 0;
    
    const float xi = particle[i].x;
    const float yi = particle[i].y;
    const float zi = particle[i].z;
    
    for (int j = 0; j < nParticles; j++) {
        // Newton's law of universal gravity
        const float dx = particle[j].x - xi;
        const float dy = particle[j].y - yi;
        const float dz = particle[j].z - zi;
        
        const float drPower32 = pow(drSquared, 3.0/2.0);
        
        const float drPower32Inv = 1.0f / drPower32;
        // Calculate the net force
        Fx += dx * G * drPower32Inv;
        Fy += dy * G * drPower32Inv;
        Fz += dz * G * drPower32Inv;
    }
    
    // Accelerate particles in response to the gravitational force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
}

\[
\vec{F}_{ij} = \frac{G m_i m_j}{|\vec{r}_j - \vec{r}_i|^3} (\vec{r}_j - \vec{r}_i)
\]

\[
\vec{F} = m \ddot{\vec{a}} = m \frac{d \vec{v}}{dt} = m \frac{d^2 \vec{x}}{dt^2}
\]

The example code assumes m=1 for all particles.
Hands-on: Explore Survey Analysis

Windows 1

• Log into Adroit
  • `ssh -l <user> adroit.princeton.edu`

• Load environment modules
  • `module load intel`

• Compile the code
  • `icpc -g -O2 -xhost -qopt-zmm-usage=high -qopenmp nbody.cpp -o nbody.out`

• Run the provided script to submit a Advisor wrapped job to the scheduler
  • `./submit_to_scheduler`

Windows 2

• Log into Adroit with X11 forwarding
  • `ssh -Y -C <user>@adroit.princeton.edu`

• Will need local xserver (XQuartz for OSX, Xming for Windows)

• Load environment modules
  • `module load intel intel-advisor`

• Open the resulting directory with Intel Advisor
  • `advixe-gui nbody-advisor`

• Click “Show My Result”

• Explore “Survey” report
Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Confirm inefficient memory access patterns

There is no confirmation that inefficient memory access patterns are present. To fix: Run a Memory Access Patterns analysis.

Data type conversions present

There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.

Use the smallest data type

The source loop contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.
```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
};

for (int i = 0; i < nParticles; i++) {
    // Components of the gravity force on particle i
    float Fx = 0, Fy = 0, Fz = 0;
    const float xi = particle[i].x;
    const float yi = particle[i].y;
    const float zi = particle[i].z;
    for (int j = 0; j < nParticles; j++) {
        // Newton's law of universal gravity
        const float dx = particle[j].x - xi;
        const float dy = particle[j].y - yi;
        const float dz = particle[j].z - zi;
        const float drPower32 = pow(drSquared, 3.0/2.0);
        const float drPower32Inv = 1.0f / drPower32;
        // Calculate the net force
        Fx += dx * G * drPower32Inv;
        Fy += dy * G * drPower32Inv;
        Fz += dz * G * drPower32Inv;
    }
    // Accelerate particles in response to the gravitational force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
}
```
Re-run Survey Analysis

Windows 1

- Compile the code
  - `icpc -g -O2 -xhost -qopt-zmm-usage=high -qopenmp -DSoA -DNo_FP_Conv nbody.cpp -o nbody.out`
- Re-run the provided script to submit a Advisor wrapped job to the scheduler
  - `/submit_to_scheduler`

Windows 2

- Re-open the resulting directory with Intel Advisor
  - `advixe-gui nbody-advisor`
- Click “Show My Result”
- Explore “Survey” report
Any Remaining Performance Issue?

Follow up: try add \texttt{-DAligned} to the compiler flag and check the result with Advisor.
Create Snapshot for Comparison

Vectorization Advisor

Vectorization Advisor

Program(s)

- Cache sources
- Cache binaries

Performance

Metrics

- Total CPU time
- Time in 1 vectorized loop
- Time in scalar code

Vectorization Gain/Efficiency

- Vectorized Loops Gain/Efficiency
- Program Approximate Gain

OP/S and Bandwidth
Roofline Comparison

- Can you make a roofline chart for the original code and the optimized one?
Intel VTune
Intel VTune Amplifier

- Accurate data
  - Hotspot
  - Processor microarchitecture
  - Memory access
  - Threading
  - I/O
- Flexible
  - Linux, Windows and Mac OS analysis GUI
  - Link data to source code and assembly
  - Easy set-up, no special compiles
- Shared memory only
  - Serial
  - OpenMP
  - MPI on a single node
A Rich Set of Predefined Analysis Types

- **Hotspots**: what functions use most time?
- **Microarchitecture Exploration**: hardware-level performance data
- **Memory Access**: identify memory-related issues
- **HPC Performance Characterization**: overview of CPU, memory and FPU utilization
- **Threading**: Identify potential parallelization opportunities/issues
Hotspots
Microarchitecture Exploration
Threading

**Elapsed Time**: 3.270s

**Effective CPU Utilization**: 11.4% (3.640 out of 32 logical CPUs)

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU utilization value.

---

**OpenMP Analysis, Collection Time**: 3.270

- **Serial Time**: 2.072s (60.4%)  
- **Parallel Region Time**: 3.189s (57.0%)  
- **Estimated Ideal Time**: 1.854s (56.7%)  
- **OpenMP Potential Gain**: 1.345s (44.2%)  

**Top OpenMP Regions by Potential Gain**

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

**Total Thread Count**: 4

**Top Functions by Inactive Wait Time with Poor CPU Utilization**

This section lists the functions sorted by the time spent waiting on synchronization or thread preemption with poor CPU Utilization.
Suggest Next Steps

• 1. L2 and L3 cache issue: try blocking technique
• 2. Thread load imbalance: try “#pragma omp parallel for schedule(dynamic)” for the outer most loop
• 3. Vectorization: try “#pragma omp simd” for the inner most loop
References

• “Compiling and Tuning for Performance using Intel Advanced Vector Extensions 512”, SC18, Intel Speakership Tutorial, Carlos Rosales-Fernandez
• “Vector Parallelism on Multi-Core Processors”, CoDas-HEP 2019, Steve Lantz