Performance and Vectorization

Jonathan Halverson, Ph.D.
Research Software and Computing Training Lead
Princeton Institute for Computational Science and Engineering
Princeton University

Fall Break 2023 Bootcamp, Monday, October 16

Acknowledgments: Bei Wang (NVIDIA) and Abhishek Biswas (Research Computing & Molecular Biology)
Outline

Part 1 (slide 3 – 17) 45 min.
- Microprocessor Evolution
- FLOPS calculations
- Basics of Vectorization
- Intel Compiler Options
- N-Body Problem

Part 2 (slide 18 – 30) 45 min.
- Vectorization Reports
- Performance Improvements
- Data Access Patterns
- Roofline Analysis Model
- Intel Advisor Demo
Single Core Performance Stalled

- Multicore
- Vectorization
- Pipelining
- Superscalar

https://github.com/karlrupp/microprocessor-trend-data
Peak FLOPS

Floating Point Operations Per Second
(Theoretically)

\[ \text{sockets} \cdot \frac{\text{cores}}{\text{socket}} \cdot \text{frequency} \cdot \text{FLOPs/cycle/core} \]

Lookup using `lscpu` on the machine:
1. Socket(s)
2. Core(s) per socket
3. CPU MHz

Lookup from manual
Vector Registers and Instructions

- 32 bit float
- 64 bit double

SSE (Streaming SIMD Extensions), 128-bit

AVX (Advanced Vector Extensions), 256-bit

AVX, 512-bit
Vector Registers and Instructions

https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
Fused Multiply-Add

Before 1999, the vector addition to the right required 4 instructions:

\[
\begin{align*}
    v3.x &= v1.x + v2.x \\
    v3.y &= v1.y + v2.y \\
    v3.z &= v1.z + v2.z \\
    v3.w &= v1.w + v2.w
\end{align*}
\]

With 128-bit vectorization, only 1 instruction is needed:

```
movaps xmm0, [v1] ; xmm0 = v1.w | v1.z | v1.y | v1.x
addps xmm0, [v2] ; xmm0 = v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
movaps [vec_res], xmm0 ; xmm0
```

FLOPs / cycle / core

- **Floating Point Operations**
  \[
  \frac{\text{instructions}}{\text{cycle}} \cdot \frac{\text{operations}}{\text{instructions}} \cdot \text{FLOPs}
  \]

- **Xeon Gold 6226R CPU @ 2.90GHz**
  - Cascade Lake / Intel Xeon 2nd Generation
  - 2 FMA (Fused Multiply Add) instructions
  - 512 AVX (Advanced Vector Extensions)

- **Double Precision FLOPs**
  - 32 DP FLOPs/cycle: 2 (VPUs/core) x 8 (8-wide) x 2 (FMA) FLOPs/element

- **Single Precision FLOPs**
  - 64 SP FLOPs/cycle: 2 (VPUs/core) x 16 (16-wide) x 2 (FMA) FLOPs/element

\[
\text{FMA} = a \times b + c
\]
Find the Peak FLOPS

- Log in to Adroit: $ ssh <YourNetID>@adroit.princeton.edu

- Find the CPU information: $ lscpu

- What is the frequency or clock speed of the chip?
  - Is the clock speed is fixed?
What is the peak FLOPS?

- **Double Precision (64 bit) FLOPS**
  
  \[
  2 \text{ sockets} \times 16 \text{ cores} \times 2.9 \times 10^9 \text{ billion cycles/second} \times 2 \text{ VPUs} \times 8 \text{ elements} \times 2 \text{ operations/element} 
  \]

  \[
  = 2970 \text{ GFLOPS} 
  \]

- **Single Precision (64 bit) FLOPS**
  
  \[
  2 \text{ sockets} \times 16 \text{ cores} \times 2.9 \times 10^9 \text{ billion cycles/second} \times 2 \text{ VPUs} \times 16 \text{ elements} \times 2 \text{ operations/element} 
  \]

  \[
  = 5940 \text{ GFLOPS} 
  \]
Three Ways to Vectorize

- **Hardware**
  - Vector instructions
  - Special registers and vector processing units

- **Software**
  - Vendor supplied libraries like Intel Math Kernel Library, BLAS etc.
  - Compiler-based
    - Automatic or with user directives

- **User Developed**
  - Special data structuring
  - Cache optimization strategies
Vectorization: Single Instruction, Multiple Data (SIMD)

- Trying to group operations

- Line up your data to operate

```c
//Sum 2 arrays x and y of length N
//Assume N is power of 2
//Unrolled

for(int i=0; i<N/4; i+=4)
    z[i+0] = x[i+0] + y[i+0]
    z[i+1] = x[i+1] + y[i+1]
    z[i+2] = x[i+2] + y[i+2]
    z[i+3] = x[i+3] + y[i+3]
```

Unroll the loop a bit to utilize the vector instructions!
Vectorizable Loops

- Number of iterations is known on entry
- Single control flow; no “if” or “switch” statements
- Must be the innermost loop, if nested
- No function calls but basic math: pow(), sqrt(), sin(), etc.
- All loop iterations must be independent of each other
Compiler Options for Vectorization

- **GCC vectorize option:** `-O2 -ftree-vectorize` or `-O3`
  - Default for x86_64 is SSE (see output from `gcc -v`)
  - To tune vectors to the host machine: `-march=native`
  - To optimize across objects (e.g., to inline): `-flto`
  - For AVX-512: `-mprefer-vector-width=512`

- **Intel Classic Compilers vectorize option:** `-O2`
  - Default is SSE instructions, 128-bit vector width (4 floats)
  - To tune vectors to the host machine: `-xHost`
  - To optimize across objects (e.g., to inline functions): `-ipo`
  - For AVX-512 in Skylake: `-qopt-zmm-usage=high`
Compiler Options for Vectorization 2

● GCC compilers for specific instruction sets: -m<feature>
  ○ Compile for AVX2: -mavx2 -mfma or -march=haswell
  ○ GCC 4.9+ has specific options for most AVX-512 extensions
  ○ GCC 5.3+ has -march=skylake-avx512
  ○ GCC 8.1+ has -march=icelake-server
  ○ GCC 9.1+ has -march=cascadelake

● Intel Classic compilers for specific instruction sets: -x<feature>
  ○ Compile for AVX2: -xCORE-AVX2
  ○ For SKL-SP and later: -xCORE-AVX512
  ○ For Xeon + KNL: -xCOMMON-AVX512 (pure KNL is -xKNL)
Connect and Download the Code

- Log in to Adroit

  ```
  ssh <YourNetID>@adroit.princeton.edu
  ```

- Clone the repo

  ```
  git clone https://github.com/jdh4/N-Body-Prob
  ```

- Work through the exercises

  ```
  https://github.com/jdh4/N-Body-Prob/blob/master/README.md
  ```
N-Body Problem

Predicting the individual motions of a group of objects interacting gravitationally

```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;

    // Loop over particles that exert force: vectorization expected here
    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 drSquared = dx*dx + dy*dy + dz*dz + softening; // 6flops
        const float drPower32 = pow(drSquared, 3.0/2.0);
        const float drPower32Inv = 1.0f / drPower32; // ldidtendent
        // Calculate the net force
        Fx += dx * G * drPower32Inv; // 3flops
        Fy += dy * G * drPower32Inv; // 3flops
        Fz += dz * G * drPower32Inv; // 3flops
    }

    // Accelerate particles in response to the gravitational force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
```
Basic Steps to Compile and Run the Code

1. Edit “ICXXFLAGS” in Makefile: `ICXXFLAGS=-O0`
2. Compile the code: `make app-ICC`
3. Check the vectorization report: `vec.report`
4. Run the code: `./app-ICC`
5. Record the performance number
6. Clean the built executable: `make clean`
Try Different Compiler Options

- **ICXXFLAGS=-O0**
  - Scalar with no vectorization
- **ICXXFLAGS=-O2**
  - 128 bit register width
- **ICXXFLAGS=-O2 -xCORE-AVX2**
  - 256 bit register width
- **ICXXFLAGS=-O2 -xCORE-AVX512 -qopt-zmm-usage=high**
  - 512 bit register width
Vectorization Reports
Vectorization Reports

The “scalar cost” means “cost of one iteration of scalar loop”.

The “vector cost” means “cost of one iteration of vectorized loop divided by vector_length*unroll_factor”, i.e. cost of somewhat equivalent to one scalar iteration.

The “vectorization overhead” shows normalized (by vector iteration cost) cost of vector initializations/finalizations before/after the loop.

The “estimated potential speedup” is calculated for the whole loop execution. It shows the normalized (by scalar iteration cost) potential gain of vectorized loop execution – including peel, remainder, and main loop for the estimated loop trip count. It can’t be derived explicitly from the scalar and vector cost shown above.
## Vectorization Reports

<table>
<thead>
<tr>
<th>Compiler Options</th>
<th>Performance (GFLOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O0</code></td>
<td>0.7</td>
</tr>
<tr>
<td><code>-O2</code></td>
<td></td>
</tr>
<tr>
<td><code>-O2 -xCORE-AVX2</code></td>
<td></td>
</tr>
<tr>
<td><code>-xCORE-AVX512 -qopt-zmm-usage=high</code></td>
<td></td>
</tr>
<tr>
<td><code>-O2 -xCORE-AVX512 -qopt-zmm-usage=high -DNo_FP_Conv</code></td>
<td></td>
</tr>
<tr>
<td><code>-O2 -xCORE-AVX512 -qopt-zmm-usage=high -DNo_FP_Conv -DSoA</code></td>
<td></td>
</tr>
<tr>
<td><code>-O2 -xCORE-AVX512 -qopt-zmm-usage=high -DNo_FP_Conv -DSoA -DOMP_SIMD -DAligned</code></td>
<td></td>
</tr>
</tbody>
</table>

Maximum theoretical performance: 1 core x 2.9 GHz x 2 VPUs x 16 elements x 2 operations/element = 185 GFLOPS
Removing FP Conversion (add: –DNo_FP_Conv)
Data Access Issues: Stride is 6

Use Structure of Array (SoA) vs. Array of Structure (AoS) data layout

```
struct Particle {
    float x, y, z;
    float vx, vy, vz;
};

Particle* particle = new Particle[nParticles];
```
Data Access Issues: Stride is 6

Use Structure of Array (SoA) vs. Array of Structure (AoS) data layout

vector register
Data Access Issues: Refactor the code

```c
struct ParticleArrays {
    float *x, *y, *z;
    float *vx, *vy, *vz;
};

ParticleArrays particle;
particle.x = new float[nParticles];
particle.y = new float[nParticles];
particle.z = new float[nParticles];
particle.vx = new float[nParticles];
particle.vy = new float[nParticles];
particle.vz = new float[nParticles];
```
Data Access Report (add: –DSoA)
Data Alignment: Peel Loop

- Data alignment with certain byte boundary
  - Align the base pointer where the space is allocated for the array
- Replace malloc() and free() with alignment specified replacement, e.g., _mm_malloc() and _mm_free() (Intel), posix_memalign() and free() etc
- Use pragmas/directives and clauses to tell the compiler that memory accesses are aligned
  - #pragma vector aligned (Intel)
  - #pragma omp simd aligned (OpenMP)
Data Alignment (add: -DOMP_SIMD -DAligned)
Roofline Analysis Model

Roofline Analysis Model Inference

Maximum Performance GFLOPS

Operational (Arithmetic) Intensity (FLOPs/byte)

Limited By Bandwidth
Loop Unrolling Parallelization SIMD
Limited By Computation

Memory Roof
Unobtainable
Compute Roof
Roofline Analysis Model Inference

Maximum Performance GFLOPS

Operational (Arithmetic) Intensity (FLOPs/byte)

- Unobtainable
- Compute Roof
- Memory Roof
- Limited By Bandwidth
  - Optimizing Cache
  - Algorithm Re-Design
- Limited By Computation
Generating Intel Advisor Report (Option 1)

- Enable 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/19.1 intel-advisor/oneapi

- Run the provided script to submit an Advisor wrapped job to the scheduler
  - bash ./run_via_slurm.sh

- Open the resulting directory with Intel Advisor
  - advixe-gui nbody-advisor
Generating Intel Advisor Report (Option 2)

- Use a graphical desktop on MyAdroit:

https://github.com/jdh4/N-Body-Prob/tree/master#roofline-analysis
N-Body Roofline Plot
Questions?