Performance Modeling and Tweaking with the Roofline Model

Case Study in Matrix-Matrix Multiplication

Felix Schwinger

High-Performance and Automatic Computing
July 06, 2016
Agenda

Introduction
   Performance Models
   Motivation

Roofline Model
   Creating a Roofline Model
   Performance Tuning Using the Roofline Model

Conclusion
Agenda

Introduction
   Performance Models
   Motivation

Roofline Model
   Creating a Roofline Model
   Performance Tuning Using the Roofline Model

Conclusion
Performance Models

- Simplify the task of understanding and improving performance of an application
Performance Models

- Simplify the task of understanding and improving performance of an application
- The model should accurately predict the program’s performance
Performance Models

- Simplify the task of understanding and improving performance of an application
- The model should accurately predict the program’s performance
- The model should be easy to use
Performance Models

- Simplify the task of understanding and improving performance of an application
- The model should accurately predict the program’s performance
- The model should be easy to use
- Possible tweaks should be shown by the model
Motivation

- Performance and scalability can be non-intuitive for new programmers
Motivation

- Performance and scalability can be non-intuitive for new programmers
- Different levels of parallelism need to be exploited on new architectures (Task-Level Parallelism, Instruction Level Parallelism)
Motivation

- Performance and scalability can be non-intuitive for new programmers
- Different levels of parallelism need to be exploited on new architectures (Task-Level Parallelism, Instruction Level Parallelism)
- Helps to evaluate if a given change to a system or application offers performance benefits before implementing it
Agenda

Introduction
  Performance Models
  Motivation

Roofline Model
  Creating a Roofline Model
  Performance Tuning Using the Roofline Model

Conclusion
Roofline Model

- Provides a graph depicting performance expectations
Roofline Model

- Provides a graph depicting performance expectations
- Shows hardware performance limitations for a given kernel
Roofline Model

- Provides a graph depicting performance expectations
- Shows hardware performance limitations for a given kernel
- Shows the benefit of a few optimizations
Audience

- Novice programmers just starting to write parallel kernels
Audience

- Novice programmers just starting to write parallel kernels
- Not for people interested in fine tuning, only gives general advices
Roofline Model

\[ P = \min(P_{\text{peak}}, I \times b_s) \]
Roofline Model

\[ P = \min(P_{\text{peak}}, I \times b_s) \]

- \( P_{\text{peak}} \) is the peak performance of the processor (Flop/s)
- \( b_s \) is the peak bandwidth of the architecture (Bytes/s)
- \( I \) is the computational intensity of the kernel (Flops/Byte)
Roofline Model

\[ P = \min(P_{\text{peak}}, I \times b_s) \]

- \( P_{\text{peak}} \) is the peak performance of the processor (F/s)
- \( b_s \) is peak bandwidth of the architecture (B/s)
Roofline Model

$$P = \min(P_{\text{peak}}, I \times b_s)$$

- $P_{\text{peak}}$ is the peak performance of the processor (F/s)
- $b_s$ is peak bandwidth of the architecture (B/s)
- $I$ is the computational intensity of the kernel (F/B)
How to obtain those values?

- Calculate Peak Performance as product of:
  - Cycles per second
  - Number of cores
  - Instructions per cycle
  - Flops per instruction
How to obtain those values?

- Calculate Peak Performance as product of:
  - Cycles per second
  - Number of cores
  - Instructions per cycle
  - Flops per instruction

- Measure memory bandwidth with the STREAM benchmark
How to obtain those values?

- Calculate Peak Performance as product of:
  - Cycles per second
  - Number of cores
  - Instructions per cycle
  - Flops per instruction

- Measure memory bandwidth with the STREAM benchmark

- Calculate Operational Intensity from the algorithm
# Hardware Used

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU Model</th>
<th>Clock Speed</th>
<th>Cores</th>
<th>SIMD</th>
<th>FMA</th>
<th>Bandwidth</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-S</td>
<td>Intel Westmere X5675</td>
<td>3.07 GHz</td>
<td>2x6</td>
<td>SSE</td>
<td>No</td>
<td>40 GB/s</td>
<td>147.36 GFlops</td>
</tr>
<tr>
<td>Home</td>
<td>Intel Haswell E3-1220v3</td>
<td>3.10 GHz</td>
<td>1x4</td>
<td>AVX2</td>
<td>Yes</td>
<td>16 GB/s</td>
<td>198.4 GFlops</td>
</tr>
</tbody>
</table>

Peak Flops = cycles per second \* number of cores \* flops per instruction \* instruction per cycle

MPI-S: 3.07 GHz \* 12 cores \* 1 \* 4 (2-wide SSE2 addition + 2-wide SSE2 multiplication) = 147.36 GFlops

Home: 3.10 GHz \* 4 cores \* 2 \* 8 (2 4-wide FMA instructions) = 198.4 GFlops
# Hardware Used

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU Model</th>
<th>Clock Speed</th>
<th>Cores</th>
<th>SIMD</th>
<th>FMA</th>
<th>Bandwidth</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-S</td>
<td>Intel Westmere X5675</td>
<td>3.07 GHz</td>
<td>2x6</td>
<td>SSE</td>
<td>No</td>
<td>40 GB/s</td>
<td>147.36 GFlops</td>
</tr>
<tr>
<td>Home</td>
<td>Intel Haswell E3-1220v3</td>
<td>3.10 GHz</td>
<td>1x4</td>
<td>AVX2</td>
<td>Yes</td>
<td>16 GB/s</td>
<td>198.4 GFlops</td>
</tr>
</tbody>
</table>

- **Peak Flops** = cycles per second $\times$ number of cores $\times$ flops per instruction $\times$ instruction per cycle
# Hardware Used

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU Model</th>
<th>Clock Speed</th>
<th>Cores</th>
<th>SIMD</th>
<th>FMA</th>
<th>Bandwidth</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-S</td>
<td>Intel Westmere X5675</td>
<td>3.07 GHz</td>
<td>2x6</td>
<td>SSE</td>
<td>No</td>
<td>40 GB/s</td>
<td>147.36 GFlops</td>
</tr>
<tr>
<td>Home</td>
<td>Intel Haswell E3-1220v3</td>
<td>3.10 GHz</td>
<td>1x4</td>
<td>AVX2</td>
<td>Yes</td>
<td>16 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

- Peak Flops = cycles per second * number of cores * flops per instruction * instruction per cycle
- MPI-S: 3.07 GHz * 12 cores * 1 * 4 (2-wide SSE2 addition + 2-wide SSE2 multiplication) = 147.36 GFlops
## Hardware Used

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU Model</th>
<th>Clock Speed</th>
<th>Cores</th>
<th>SIMD</th>
<th>FMA</th>
<th>Bandwidth</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-S</td>
<td>Intel Westmere X5675</td>
<td>3.07 GHz</td>
<td>2x6</td>
<td>SSE</td>
<td>No</td>
<td>40 GB/s</td>
<td>147.36 GFlops</td>
</tr>
<tr>
<td>Home</td>
<td>Intel Haswell E3-1220v3</td>
<td>3.10 GHz</td>
<td>1x4</td>
<td>AVX2</td>
<td>Yes</td>
<td>16 GB/s</td>
<td>198.4 GFlops</td>
</tr>
</tbody>
</table>

- Peak Flops = cycles per second * number of cores * flops per instruction * instruction per cycle
- MPI-S: 3.07 GHz * 12 cores * 1 * 4 (2-wide SSE2 addition + 2-wide SSE2 multiplication) = 147.36 GFlops
- Home: 3.10 GHz * 4 cores * 2 * 8 (2 4-wide FMA instructions) = 198.4 GFlops
Naive Matrix-Matrix Multiplication

```c
for (i=0; i<N; i++)
    for(j=0; j<N; j++)
        for (k=0; k<N; k++)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

- Operational Intensity $I = \frac{f}{m}$
  - $f$: # of floating-point adds and multiplies
  - $m$: # of words moved between memory and cache
Naive Matrix-Matrix Multiplication

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

- Operational Intensity $I = \frac{f}{m}$
  - $f$: # of floating-point adds and multiplies
  - $m$: # of words moved between memory and cache

- Assume there is enough room for a few matrix rows of size $N$
- Not enough memory for all $N^2$ elements of a matrix
Naive Matrix-Matrix Multiplication

```plaintext
for (i=0; i<N; i++)
  for(j=0; j<N; j++)
    for (k=0; k<N; k++)
      res[i][j] += mul1[i][k] * mul2[k][j];
```

- Operational Intensity $I = \frac{f}{m} \approx \frac{2*N^3}{1*N^3} = 2$
  - $f$: # of floating-point adds and multiplies
  - $m$: # of words moved between memory and cache
- Assume there is enough room for a few matrix rows of size $N$
- Not enough memory for all $N^2$ elements of a matrix
Naive Matrix-Matrix Multiplication

```c
for (i=0; i<N; i++)
    for(j=0; j<N; j++)
        for (k=0; k<N; k++)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

- Operational Intensity \( I = \frac{f}{m} \approx \frac{2 \cdot N^3}{1 \cdot N^3} = 2 \)
  - \( f \): # of floating-point adds and multiplies
  - \( m \): # of words moved between memory and cache

- Assume there is enough room for a few matrix rows of size \( N \)
- Not enough memory for all \( N^2 \) elements of a matrix

Horrible performance, do not use if performance is relevant

Highly optimized libraries available for linear algebra operations
Roofline Model for naive Matrix-Matrix Multiplication

Operational Intensity [Flop/Byte]
0.0625 0.125 0.25 0.5 1 2 4 8 16 32 64 128

Performance [GFlop/Second]
No FMA
No SIMD/ILP

\( I \times b \)

\( P_{\text{peak}} \)

Performance Modeling and Tweaking with the Roofline Model
Felix Schwinger | High-Performance and Automatic Computing | July 06, 2016
Tiled Matrix-Matrix Multiplication

```c
for (i = 0; i < N; i += b)
    for (j = 0; j < N; j += b)
        for (k = 0; k < N; k += b)
            for (i2 = 0; i2 < b; ++i2)
                for (k2 = 0; k2 < b; ++k2)
                    for (j2 = 0; j2 < b; ++j2)
                        res[i2][j2] += mul1[i2][k2]*mul2[k2][j2];
```

- **Operational Intensity** \( I = \frac{f}{m} \)
  - \( f \): \# of floating-point adds and multiplies
  - \( m \): \# of words moved between memory and cache
Tiled Matrix-Matrix Multiplication

```cpp
for (i = 0; i < N; i += b)
    for (j = 0; j < N; j += b)
        for (k = 0; k < N; k += b)
            for (i2 = 0; i2 < b; ++i2)
                for (k2 = 0; k2 < b; ++k2)
                    for (j2 = 0; j2 < b; ++j2)
                        res[i2][j2] += mul1[i2][k2]*mul2[k2][j2];
```

- **Operational Intensity** \( I = \frac{f}{m} \)
  - \( f \): # of floating-point adds and multiplies
  - \( m \): # of words moved between memory and cache

- Ignore cost of moving res, each block only moves twice
- Each block of mul1 and mul2 moves \( N \) times
Tiled Matrix-Matrix Multiplication

```c
for (i = 0; i < N; i += b)
    for (j = 0; j < N; j += b)
        for (k = 0; k < N; k += b)
            for (i2 = 0; i2 < b; ++i2)
                for (k2 = 0; k2 < b; ++k2)
                    for (j2 = 0; j2 < b; ++j2)
                        res[i2][j2] += mul1[i2][k2]*mul2[k2][j2];
```

- **Operational Intensity** 
  \[ I = \frac{f}{m} \approx \frac{(2*b^3)*n^3}{(2*b^2)*n^3} = b \]

  - \( f \): # of floating-point adds and multiplies
  - \( m \): # of words moved between memory and cache
  - \( n \): \( \frac{N}{b} \)

- Ignore cost of moving res, each block only moves twice
- Each block of mul1 and mul2 moves N times
Tiled Matrix-Matrix Multiplication

```c
for (i = 0; i < N; i += b)
    for (j = 0; j < N; j += b)
        for (k = 0; k < N; k += b)
            for (i2 = 0; i2 < b; ++i2)
                for (k2 = 0; k2 < b; ++k2)
                    for (j2 = 0; j2 < b; ++j2)
                        res[i2][j2] += mul1[i2][k2]*mul2[k2][j2];
```

- **Operational Intensity** $I = \frac{f}{m} \approx \frac{(2*b^3)*n^3}{(2*b^2)*n^3} = b$
  - $f$: # of floating-point adds and multiplies
  - $m$: # of words moved between memory and cache
  - $n$: $N/b$

- Ignore cost of moving res, each block only moves twice
- Each block of mul1 and mul2 moves N times
- Better performance, but still much slower than tuned libraries
**Single Instruction Multiple Data**

- SIMD allows to apply a operation to a "vector" in one cycle
Single Instruction Multiple Data

- SIMD allows to apply a operation to a "vector" in one cycle
- Vector length for SSE: 128 bit, e.g. 4 floats, 2 double
Single Instruction Multiple Data

- SIMD allows to apply an operation to a "vector" in one cycle
- Vector length for SSE: 128 bit, e.g. 4 floats, 2 double
- Vector length for AVX: 256 bit, e.g. 8 floats, 4 double
Single Instruction Multiple Data

- SIMD allows to apply a operation to a "vector" in one cycle
- Vector length for SSE: 128 bit, e.g. 4 floats, 2 double
- Vector length for AVX: 256 bit, e.g. 8 floats, 4 double
- Haswell microarchitecture allows 2 AVX instructions per cycle
SIMD Optimization

- Peak performance only attainable, if the algorithm vectorizes well
SIMD Optimization

- Peak performance only attainable, if the algorithm vectorizes well
- Without using SIMD a factor 2 for SSE and factor 4 for AVX is lost
Roofline Model SIMD Optimization

Performance Modeling and Tweaking with the Roofline Model
Felix Schwinger | High-Performance and Automatic Computing | July 06, 2016
Fused Multiply–Add

- FMA allows the operation $a \leftarrow a + (b \times c)$ in a single cycle.
Fused Multiply–Add

- FMA allows the operation $a \leftarrow a + (b \times c)$ in a single cycle
- Performs two floating point operations in a single cycle
Fused Multiply–Add

- FMA allows the operation $a \leftarrow a + (b \times c)$ in a single cycle
- Performs two floating point operations in a single cycle
- The result is only rounded once, i.e. result is more accurate
Fused Multiply–Add Optimization

- Peak performance only attainable, if algorithm uses a well balanced mix of additions and multiplications
Fused Multiply–Add Optimization

- Peak performance only attainable, if algorithm uses a well balanced mix of additions and multiplications
- If only additions or only multiplications are used, a factor 2 of performance will be lost
Roofline Model Fused Multiply-Add optimization
Agenda

Introduction
  Performance Models
  Motivation

Roofline Model
  Creating a Roofline Model
  Performance Tuning Using the Roofline Model

Conclusion
Conclusion

- Performance Models are used for simpler evaluation of a program’s performance.
Conclusion

- Performance Models are used for simpler evaluation of a program’s performance
- Creating a Roofline Model in a few simple steps
Conclusion

- Performance Models are used for simpler evaluation of a program’s performance
- Creating a Roofline Model in a few simple steps
- Compute-bound and memory-bound algorithms
Performance Models are used for simpler evaluation of a program’s performance

Creating a Roofline Model in a few simple steps

Compute-bound and memory-bound algorithms

Optimizing compute-bound code using the Roofline Model
References


Williams, Samuel, Andrew Waterman, and David Patterson. [https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/](https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/)