Case Study: Matrix Multiplication

6.S898: Advanced Performance Engineering for Multicore Applications
February 22, 2017
# 4k-by-4k Matrix Multiplication

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>25,552.48</td>
<td>0.005</td>
<td>1</td>
<td>—</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,372.68</td>
<td>0.058</td>
<td>11</td>
<td>10.8</td>
<td>0.01%</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide-and-conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>62,806</td>
<td>2.7</td>
<td>40.45%</td>
</tr>
<tr>
<td>6</td>
<td>+ vectorization</td>
<td>1.10</td>
<td>124.914</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>+ AVX intrinsics</td>
<td>0.41</td>
<td>337.812</td>
<td>6,727</td>
<td>3.5</td>
<td>40.45%</td>
</tr>
<tr>
<td>8</td>
<td>Strassen</td>
<td>0.38</td>
<td>361.177</td>
<td>67,150</td>
<td>1.1</td>
<td>43.24%</td>
</tr>
</tbody>
</table>

Today, we’ll look into the performance engineering behind versions 3–7.
Outline

- The matrix multiplication problem
- Serial and parallel looping codes
- Cache-efficient matrix multiplication
- Hands-on: Vectorization using the compiler
- Vectorization by hand
Problem: Compute the product $C = (c_{ij})$ of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$.

The matrix product obeys the following formula:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

For simplicity, we shall assume that $n$ is a power of 2.
Three Nested Loops in C

Work of computation:

- $n^3$ iterations
- Each iteration performs constant work.

$\Theta(n^3)$ total work.

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
</tbody>
</table>
Parallel Loops

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td>366</td>
<td>7.8</td>
<td>0.24%</td>
</tr>
</tbody>
</table>

But the machine has 18 cores! Where’s my 18x speedup!?
Work/Span Analysis of Parallel Loops

- **Work:** $T_1(n) = \Theta(n^3)$
- **Span:** $T_\infty(n) = \Theta(\log n + \log n + n)$
- **Parallelism:** $T_1(n)/T_\infty(n) = \Theta(n^2)$

This code has ample parallelism, but still gets poor parallel speedup!
Memory Access Pattern for Looping Code

Matrices are stored in **row-major order**.

Layout of matrices in memory:

- **C**
- **A**
- **B**
Cache Analysis of Looping Code

Suppose that $n$ is sufficiently large. Let $B$ be the size of a cache line.

❖ Computing an element of matrix $C$ involves $\Theta(n/B)$ cache misses for matrix $A$ and $\Theta(n)$ cache misses for matrix $B$.

❖ **No temporal locality** on matrix $B$. Cache can’t store all of the cache lines for one column of matrix $B$.

❖ Computing each element of matrix $C$ incurs $\Theta(n)$ cache misses.

❖ In total, $\Theta(n^3)$ cache lines are read to compute all of matrix $C$.  

Layout of matrices in memory:
Improving Cache Efficiency

We can improve cache efficiency using a recursive divide-and-conquer algorithm.

- Imagine each matrix is subdivided into four quadrants.
  
  \[
  C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} \quad A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \quad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}
  \]

- The matrix product can be expressed recursively in terms of 8 products of submatrices:
  
  \[
  \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}
  \]
Recursive Divide-And-Conquer

```c
void mmdac(double *restrict C, double *restrict A, double *restrict B, int size, int n)
{
    if (size <= THRESHOLD) {
        mmbase(C, A, B, size);
    } else {
        int s00 = 0;
        int s01 = size/2;
        int s10 = (size/2)*n;
        int s11 = (size/2)*(n+1);
        mmdac(C+s00, A+s00, B+s00, size/2, n);
        mmdac(C+s01, A+s00, B+s01, size/2, n);
        mmdac(C+s10, A+s10, B+s00, size/2, n);
        mmdac(C+s11, A+s10, B+s01, size/2, n);
        mmdac(C+s00, A+s01, B+s10, size/2, n);
        mmdac(C+s01, A+s01, B+s11, size/2, n);
        mmdac(C+s10, A+s11, B+s10, size/2, n);
        mmdac(C+s11, A+s11, B+s11, size/2, n);
    }
}
```
Analysis of Recursive Divide-And-Conquer

Work of computation:
- Recurrence:
  \[ T(n) = 8T(n/2) + \Theta(1) \]
- Solve the recurrence via the Master Method:
  \[ T(n) = \Theta(n^3) \]
Analysis of Recursive Divide-And-Conquer

Cache complexity: Let $M$ be the cache size and $B$ the size of a cache line. Assume the base case size fits in cache.

❖ Base case incurs $\Theta(n^2 / B)$ cache misses.
❖ Recursive case incurs $Q(n) = 8Q(n/2) + \Theta(1)$ cache misses.
❖ Solution: $Q(n) = \Theta(n^3 / M^{1/2}B)$

```c
void mmdac(double *restrict C, double *restrict A, double *restrict B, int size, int n) {
    if (size <= THRESHOLD) {
        mmbase(C, A, B, size);
    } else {
        int s00 = 0;
        int s01 = size/2;
        int s10 = (size/2)*n;
        int s11 = (size/2)*(n+1);
        mmdac(C+s00, A+s00, B+s00, size/2, n);
        mmdac(C+s01, A+s00, B+s01, size/2, n);
        mmdac(C+s10, A+s10, B+s00, size/2, n);
        mmdac(C+s11, A+s10, B+s01, size/2, n);
        mmdac(C+s00, A+s01, B+s10, size/2, n);
        mmdac(C+s01, A+s01, B+s11, size/2, n);
        mmdac(C+s10, A+s11, B+s10, size/2, n);
        mmdac(C+s11, A+s11, B+s11, size/2, n);
    }
}
```
void mmdac(double *restrict C, double *restrict A, double *restrict B, int size, int n) {
    if (size <= THRESHOLD) {
        mmbase(C, A, B, size);
    } else {
        int s00 = 0;
        int s01 = size/2;
        int s10 = (size/2)*n;
        int s11 = (size/2)*(n+1);
        cilk_spawn mmdac(C+s00, A+s00, B+s00, size/2, n);
        cilk_spawn mmdac(C+s01, A+s00, B+s01, size/2, n);
        cilk_spawn mmdac(C+s10, A+s10, B+s00, size/2, n);
        mmdac(C+s11, A+s10, B+s01, size/2, n);
        cilk_sync;
        cilk_spawn mmdac(C+s00, A+s01, B+s10, size/2, n);
        cilk_spawn mmdac(C+s01, A+s01, B+s11, size/2, n);
        cilk_spawn mmdac(C+s10, A+s11, B+s10, size/2, n);
        mmdac(C+s11, A+s11, B+s11, size/2, n);
        cilk_sync;
    }
}

This code has ample parallelism.

Work: 
\[ T_1(n) = \Theta(n^3) \]

Span: 

Recurrence: 
\[ T_\infty(n) = 2T_\infty(n/2) + \Theta(1) \]

Solution: 
\[ T_\infty(n) = \Theta(n) \]
## Performance of Parallel Divide-And-Conquer

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td>366</td>
<td>7.8</td>
<td>0.24%</td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide-and-conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>6,727</td>
<td>18.4</td>
<td>4.33%</td>
</tr>
</tbody>
</table>
Where To Optimize Next?

Work of computation:

- Write the recurrence:
  \[ T(n) = 8T(n/2) + \Theta(1) \]

- Solve the recurrence via Master Method:
  \[ T(n) = \Theta(n^3) \]

Practically all of the work is in the base case!
Hands-On: Implement the Base Case

- Download mm_dac.c: [http://pastebin.com/dl/MSmqi5Bq](http://pastebin.com/dl/MSmqi5Bq)
- Implement a simple base case.
- Compile the code:
  
  ```
  $ clang -O3 -g -fcilkplus -o mm_dac mm_dac.c
  ```
- Run it!
Hands-On: Vectorization Report

Is this compiler vectorizing your code?

❖ Add the flags `-Rpass=vector` and `-Rpass-analysis=vector` to your clang arguments to get a vectorization report.

❖ What does the report say?

IEEE Floating-Point Arithmetic

IEEE floating-point arithmetic is not associative.

- The statement `printf("%.17f", (0.1+0.2)+0.3);` produces 0.6000000000000009.
- The statement `printf("%.17f", 0.1+(0.2+0.3));` produces 0.59999999999999998.

The compiler must assume that you care about this imprecision and therefore cannot reorder the floating-point operations in order to vectorize.
Hands-On: Vectorization, Attempt 1

We don’t care about this level of precision in the code’s floating-point arithmetic, so let’s add the `-ffast-math` flag to `clang` command.

❖ Is the performance any better?
❖ What does the vectorization report say now?
LLVM does not deem it efficient to vectorize the innermost loop, which reads a column of matrix B.
Here are two strategies you can try for fixing this problem:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Resulting vectorizable access pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Transpose matrix B.</td>
<td>C (\times) A (\times) B(^T)</td>
</tr>
<tr>
<td>2. Interchange the loops.</td>
<td>C (\times) A (\times) B</td>
</tr>
</tbody>
</table>
Modern Intel processors support the AVX vector instruction set.

- AVX supports **256-bit vector registers**, whereas the older SSE instruction set supports 128-bit vector registers.
- Many common AVX instructions operate on 3 operands, rather than 2, making them **easier to use**.
Hands-On: Vectorization, Attempt 3

Once you have code that vectorizes, try using the AVX instructions, which can operate on 4 elements each.

❖ Add the -mavx flag to your clang command.
❖ What does the vectorizer report say now?
❖ Did you get a performance increase?
## Performance With Vectorization

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td>366</td>
<td>7.8</td>
<td>0.24%</td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide-and-conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>6,727</td>
<td>18.4</td>
<td>4.33%</td>
</tr>
<tr>
<td>6</td>
<td>+ vectorization</td>
<td>1.10</td>
<td>124.914</td>
<td>23,224</td>
<td>3.5</td>
<td>14.96%</td>
</tr>
</tbody>
</table>

How do we go even faster?
Vector Intrinsics

Intel provides a library of intrinsic instructions for accessing their various vector instruction sets.

- C/C++ header: immintrin.h
Some Useful AVX/AVX2 Instructions

If we stare at this database and think creatively, we come up with an alternative base case for matrix multiplication!

- The \texttt{__m256d} type stores a vector of 4 doubles.
- The AVX intrinsics \texttt{_mm256_add_pd()} and \texttt{_mm256_mul_pd()} perform addition and multiplication.
- The AVX2 intrinsic \texttt{_mm256_fmadd_pd()} performs a \texttt{fused multiply-add}.
- The AVX intrinsics \texttt{_mm256_permute_pd()} and \texttt{_mm256_permute2f128_pd()} permute AVX registers.
Outer Product Base Case

**Idea:** Compute outer products between subcolumns of matrix A by subrows of matrix B.

Outer product produces a submatrix of C.

Store each subcolumn or subrow in 1 vector register.

Store intermediate submatrix of C in 4 vector registers.
Computing One Outer Product

Compute 4 vector multiplications between the subcolumn of matrix $A$ and the subrow of matrix $B$. 

Vector permutations

\[
C \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} 
\]

\[
C \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} 
\]

\[
C \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} 
\]

\[
C \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} 
\]
Computing a Whole Submatrix

- Iterate through subcolumns of A and subrows of B to compute a submatrix of C.
- Accumulate elements of C submatrix in separate vector registers.
- Once done, write C submatrix back to memory.
- All operations are element-wise!
Why Is This Base Case Fast?

The whole base case can be implemented within vector registers using a few vector operations.

- 2 AVX registers to store a subcolumn of A and its permutation.
- 2 AVX registers to store a subrow of B and its permutation.
- 4 AVX registers to store a submatrix of C.
- 2 vector permutation operations.
- 4 vector multiplication and addition operations per subrow-subcolumn pair.
## 4k-by-4k Matrix Multiplication

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>25,552.48</td>
<td>0.005</td>
<td>1</td>
<td>—</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,372.68</td>
<td>0.058</td>
<td>11</td>
<td>10.8</td>
<td>0.01%</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>47</td>
<td>4.4</td>
<td>0.03%</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td>366</td>
<td>7.8</td>
<td>0.24%</td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide-and-conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>6,727</td>
<td>18.4</td>
<td>4.33%</td>
</tr>
<tr>
<td>6</td>
<td>+ vectorization</td>
<td>1.10</td>
<td>124.914</td>
<td>23,224</td>
<td>3.5</td>
<td>14.96%</td>
</tr>
<tr>
<td>7</td>
<td>+ AVX intrinsics</td>
<td>0.41</td>
<td>337.812</td>
<td>62,806</td>
<td>2.7</td>
<td>40.45%</td>
</tr>
<tr>
<td>8</td>
<td>Strassen</td>
<td>0.38</td>
<td>361.177</td>
<td>67,150</td>
<td>1.1</td>
<td>43.24%</td>
</tr>
</tbody>
</table>