NORTHWEST INSTITUTE for ADVANCED COMPUTING

Pacific Northwest NATIONAL LABORATORY

Proudly Operated by Battelle for the U.S. Department of Energy

UNIVERSITY of WASHINGTON

AMATH 483/583
High Performance Scientific Computing

Lecture 7:
Compilation, optimization, SIMD/Vector

Andrew Lumsdaine

Northwest Institute for Advanced Computing

Pacific Northwest National Laboratory

University of Washington

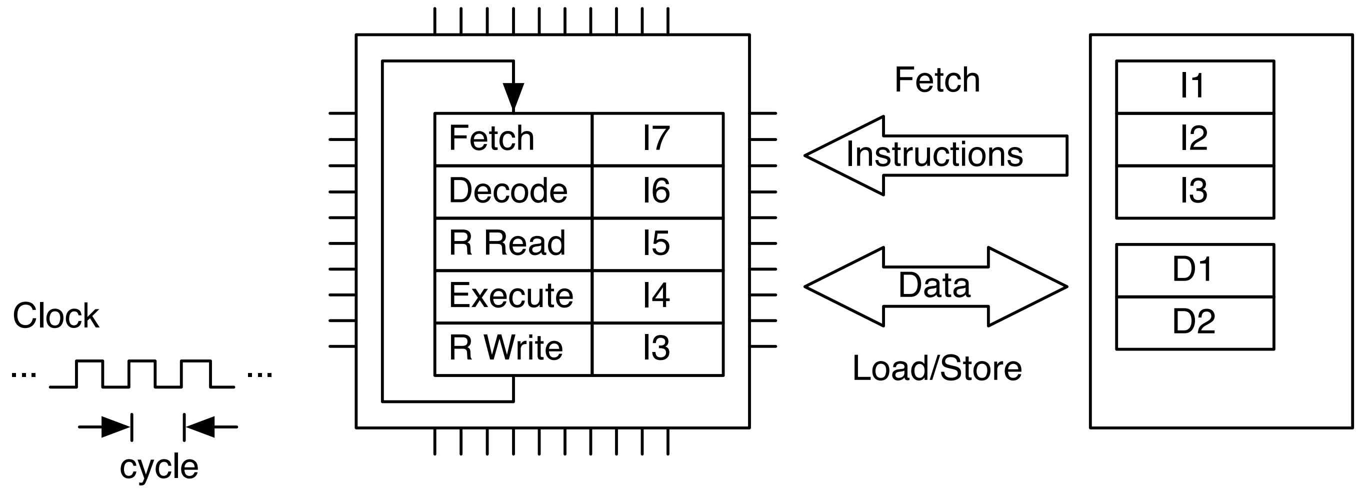
Seattle, WA

Overview

- Brief review of optimization techniques
- Doing more at once
- Vector instruction sets and intrinsics
- Sparsity

Processor Core Instruction Handling

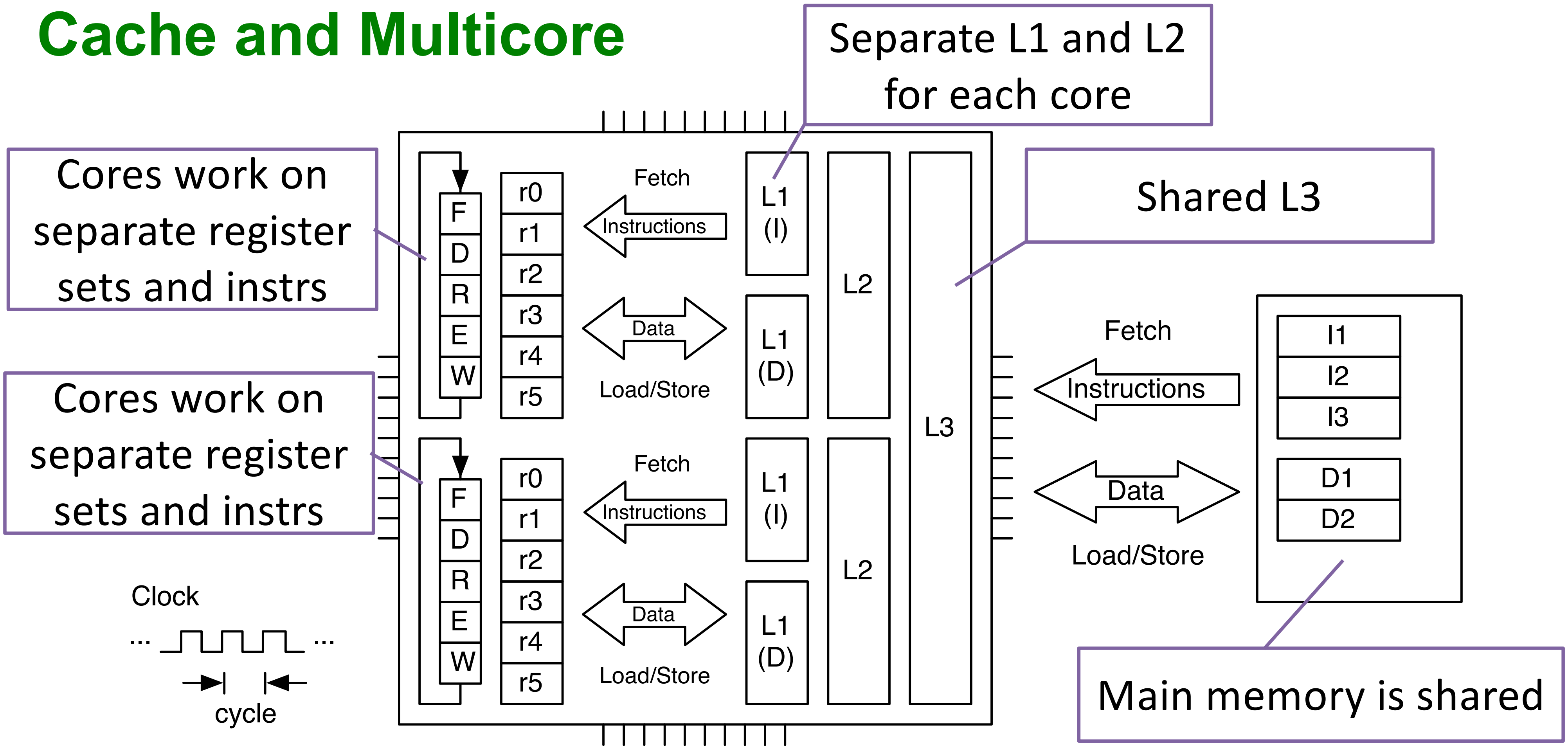
- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism (ILP)



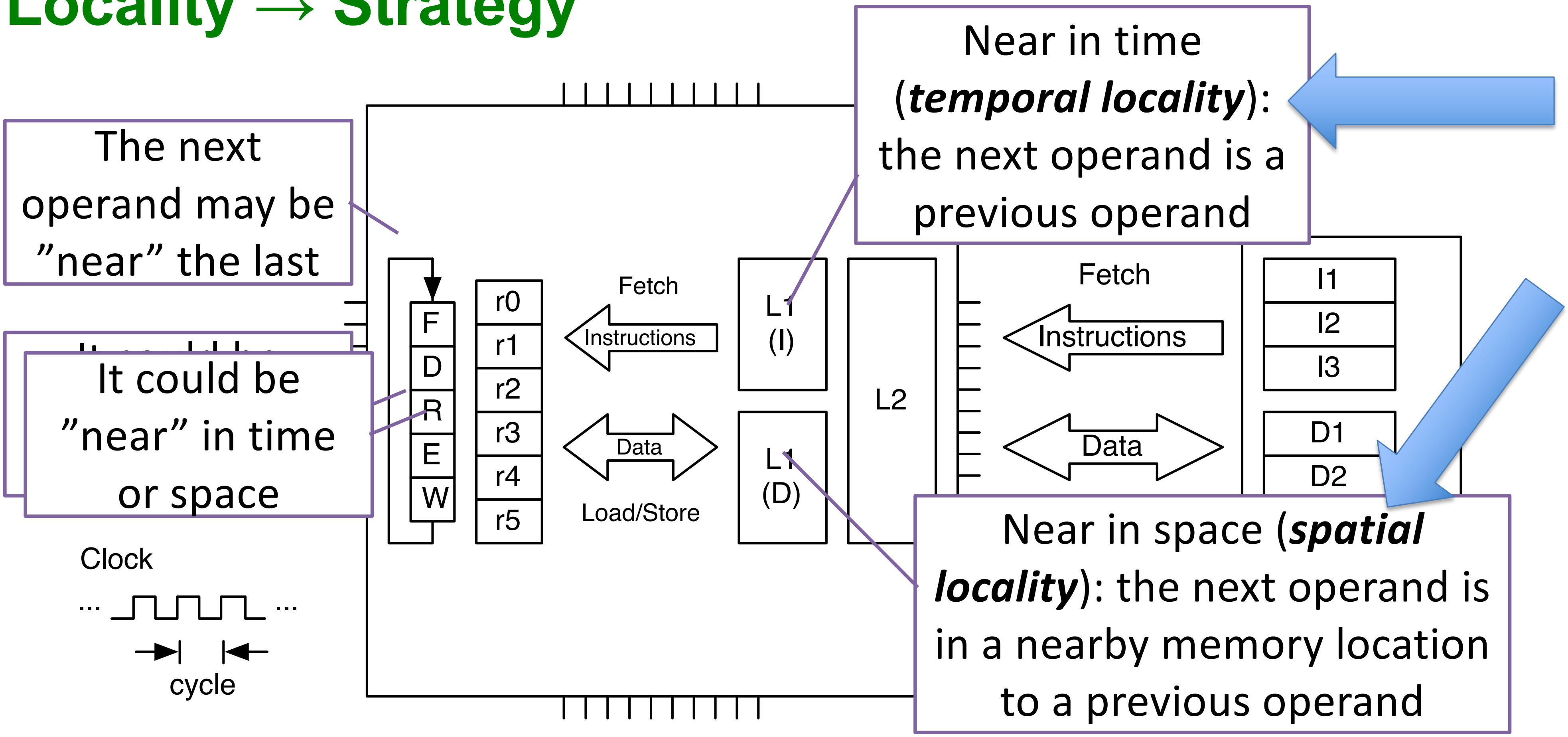
Performance-Oriented Architecture Features

- Execution Pipeline
  - Stages of functionality to process issued instructions
  - Hazards are conflicts with continued execution
  - Forwarding supports closely associated operations exhibiting precedence constraints
- Out of Order Execution
  - Uses reservation stations
  - Hides some core latencies and provide fine grain asynchronous operation supporting concurrency
- Branch Prediction
  - Permits computation to proceed at a conditional branch point prior to resolving predicate value
  - Overlaps follow-on computation with predicate resolution
  - Requires roll-back or equivalent to correct false guesses
  - Sometimes follows both paths, and several deep

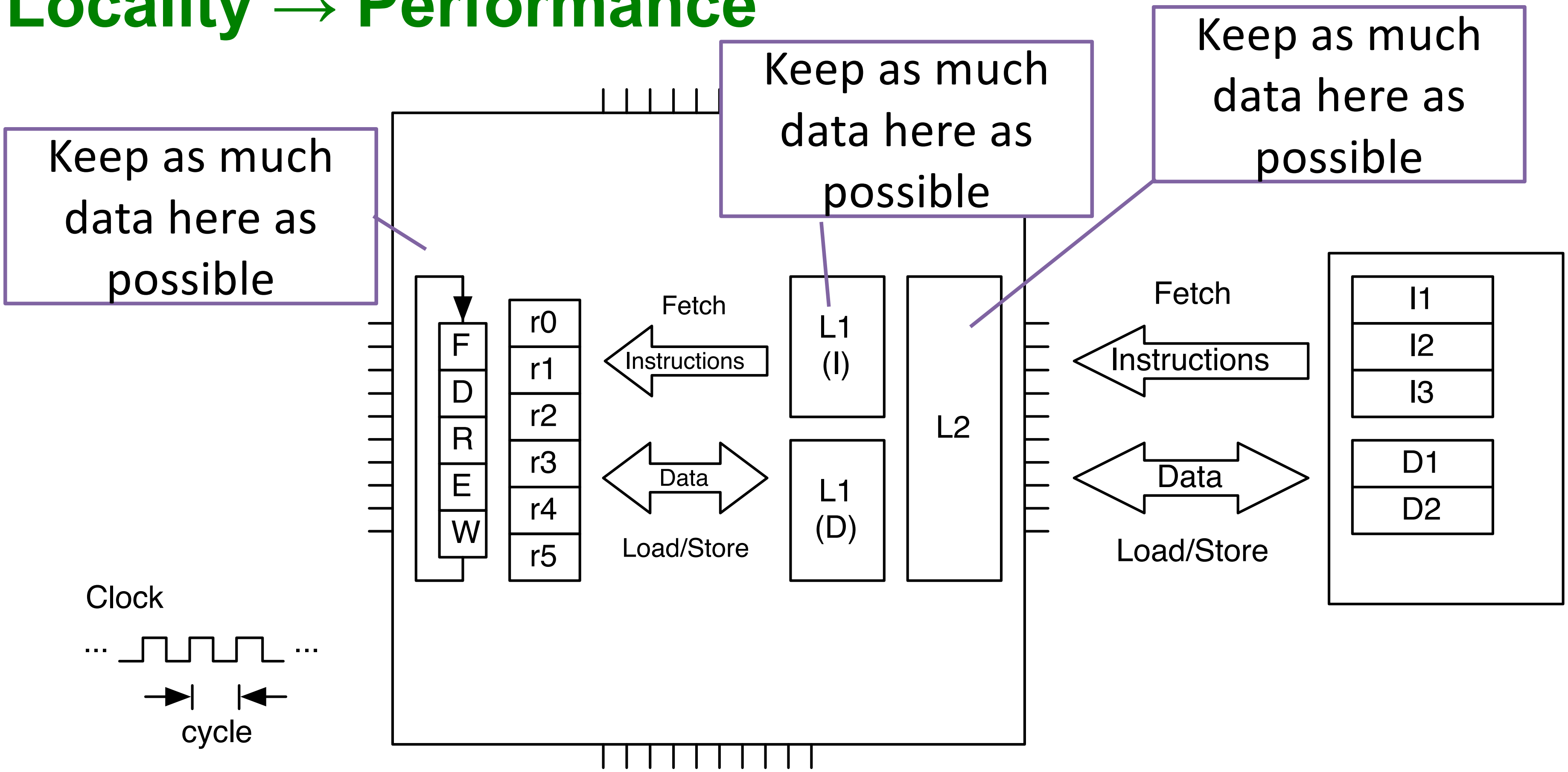
Cache and Multicore



Locality → Strategy



Locality → Performance



Our Matrix class

Matrix.hpp

```
class Matrix {
public:
  Matrix(size_t M, size_t N) : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_) {}

        double& operator()(size_t i, size_t j)       { return storage_[i * num_cols_ + j]; }
  const double& operator()(size_t i, size_t j) const { return storage_[i * num_cols_ + j]; }

  size_t num_rows() const { return num_rows_; }
  size_t num_cols() const { return num_cols_; }

private:
  size_t              num_rows_, num_cols_;
  std::vector<double> storage_;
};
```

Overloaded operator()

Expressiveness

```
Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.num_rows(), B.num_cols());
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j ) {
      for (size_t k = 0; k < A.num_cols(); ++k ) {
        C(i, j) += A(i, k) * B(k, j);
      }
    }
  }
  return C;
}
```

You can write:

```
Matrix A(5, 5), B(5, 5), C(5, 5), D(5,5);
D = A*B + C;
```

Just For Benchmarking

```
Matrix operator*(const Matrix& A, const Matrix&B) {
  Matrix C(A.num_rows(), B.num_cols());
  multiply(A, B, C);
  return C;
}

void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
```

C++ Core Guideline Violation

F.20: For "out" output values, prefer return values to output parameters

Benchmarking

```
double benchmark(size_t M, size_t N, size_t K, size_t numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (size_t i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

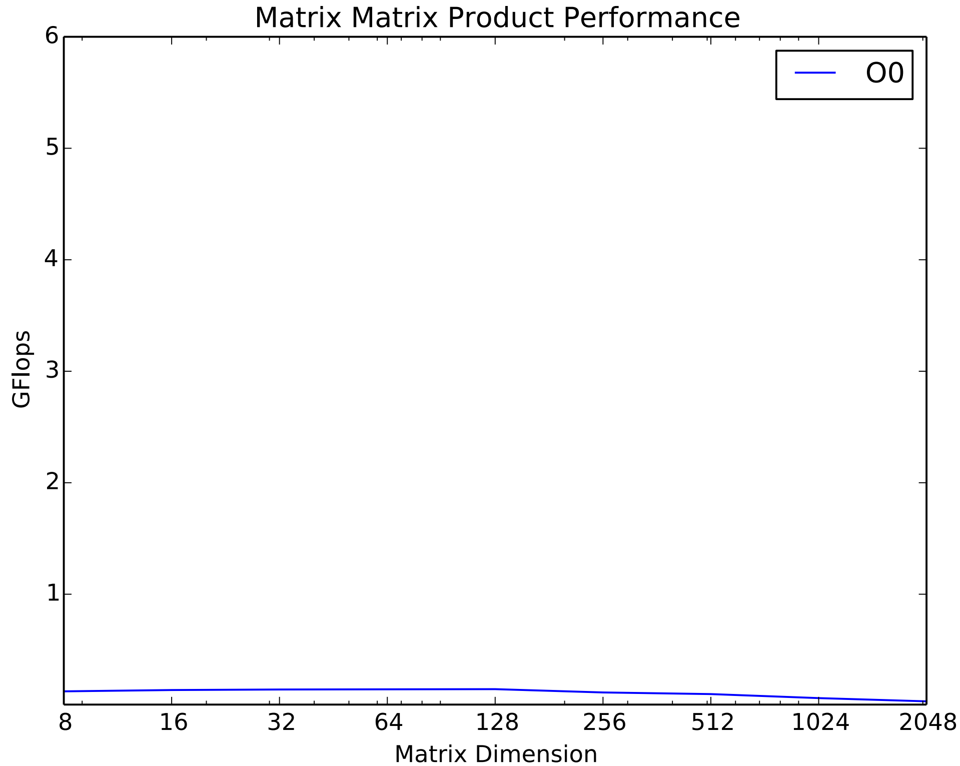
Run the core loop many times to get sufficient resolution for small(er) sizes

Let's Start Benchmarking

```
double benchmark(size_t M, size_t N, size_t K, size_t numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (size_t i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

```
bench: bench.o Matrix.o
c++ -std=c++11 bench.o Matrix.o -o bench

bench.o: bench.cpp Matrix.hpp
c++ -std=c++11 -c bench.cpp -o bench.o

Matrix.o: Matrix.cpp Matrix.hpp
c++ -std=c++11 -c Matrix.cpp -o Matrix.o
```

Base Performance Results



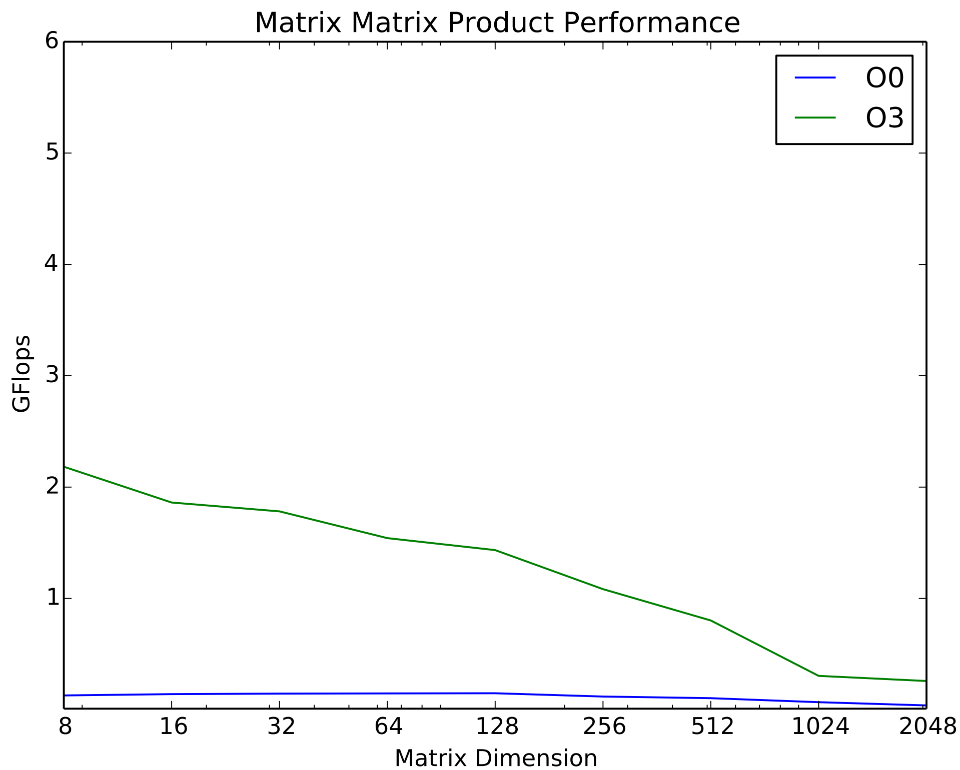
Let’s Make One Small Change

```
double benchmark(size_t M, size_t N, size_t K, size_t numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (size_t i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

Tell the compiler to use optimization level 3

```
bench: bench.o Matrix.o
c++ -O3 -std=c++11 bench.o Matrix.o -o bench

bench.o: bench.cpp Matrix.hpp
c++ -O3 -std=c++11 -c bench.cpp -o bench.o

Matrix.o: Matrix.cpp Matrix.hpp
c++ -O3 -std=c++11 -c Matrix.cpp -o Matrix.o
```

Base Performance Results



The Three Most Important Requirements for HPC

- Locality
- Locality
- Locality

Improving Locality

```
void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_ k = 0; k < A.num_cols(); ++k) {
        C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
```

What can be reused?

- Load `C(i,j)` into register
- Load `A(i,k)` into register
- Load `B(k,j)` into register
- Multiply
- Add
- Store `C(i,j)`

- Four memory operations and two floating point operations per iteration
- 2/6 = 1/3 flop per cycle (if each operation is one cycle)

Hoisting

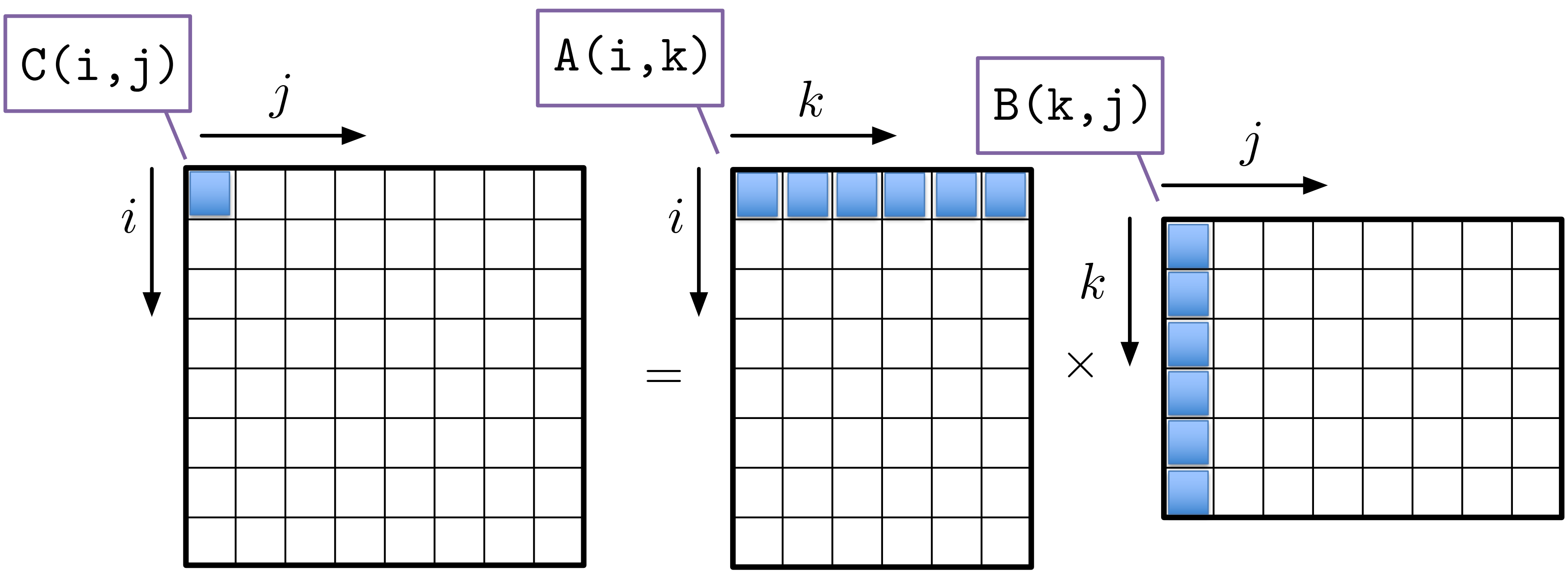
Hoist C(i,j)

```
void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      double t = C(i,j);
      for (size_t k = 0; k < A.num_cols(); ++k) {
        t += A(i,k) * B(k,j);
      }
      C(i,j) = t;
    }
  }
```

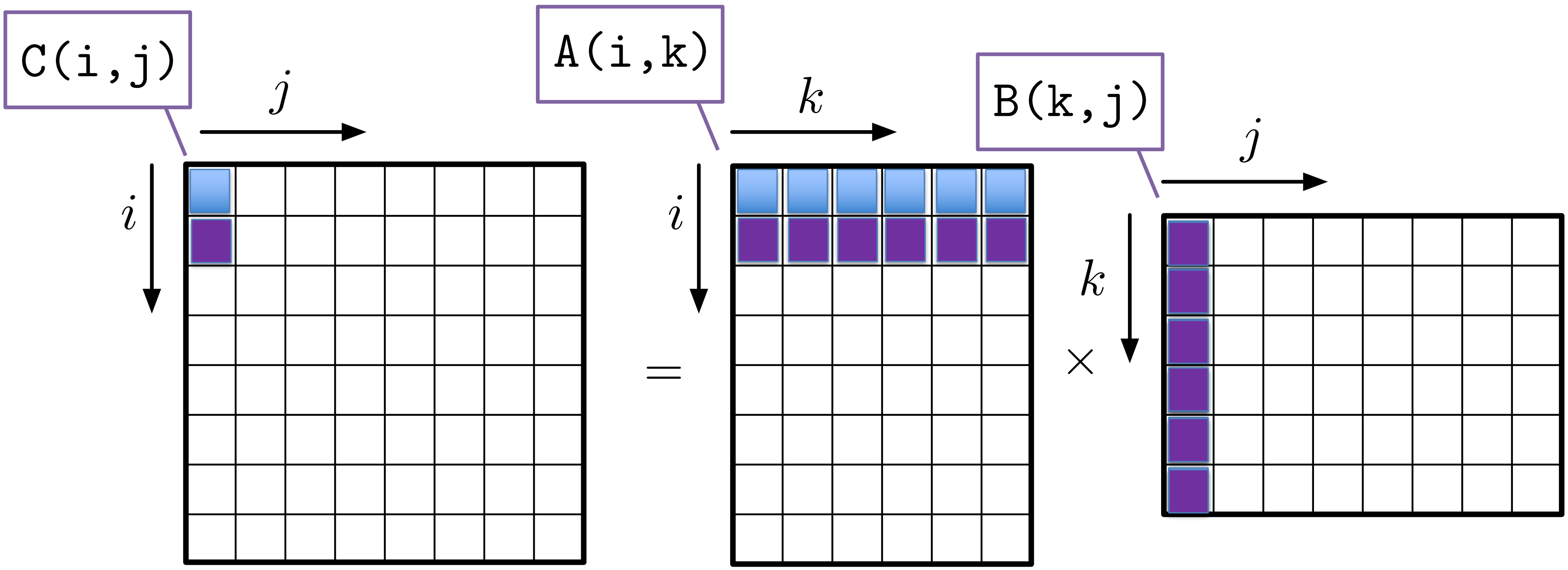
- Load `A(i,k)`
- Load `B(k,j)`
- Multiply
- Add

- Two memory operations and two floating point operations per iteration
- 2/4 = 1/2 flop per cycle (if each operation is one cycle)

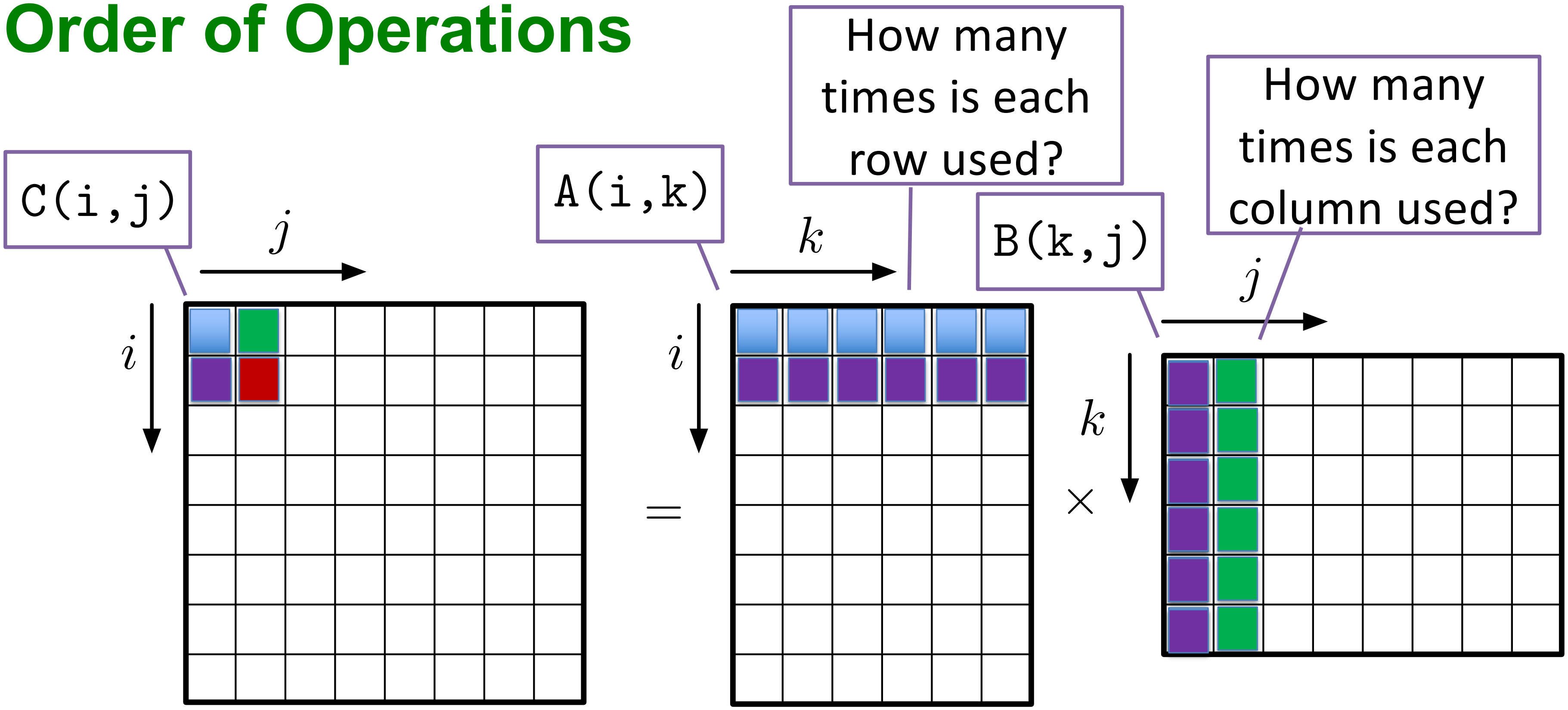
Order of Operations



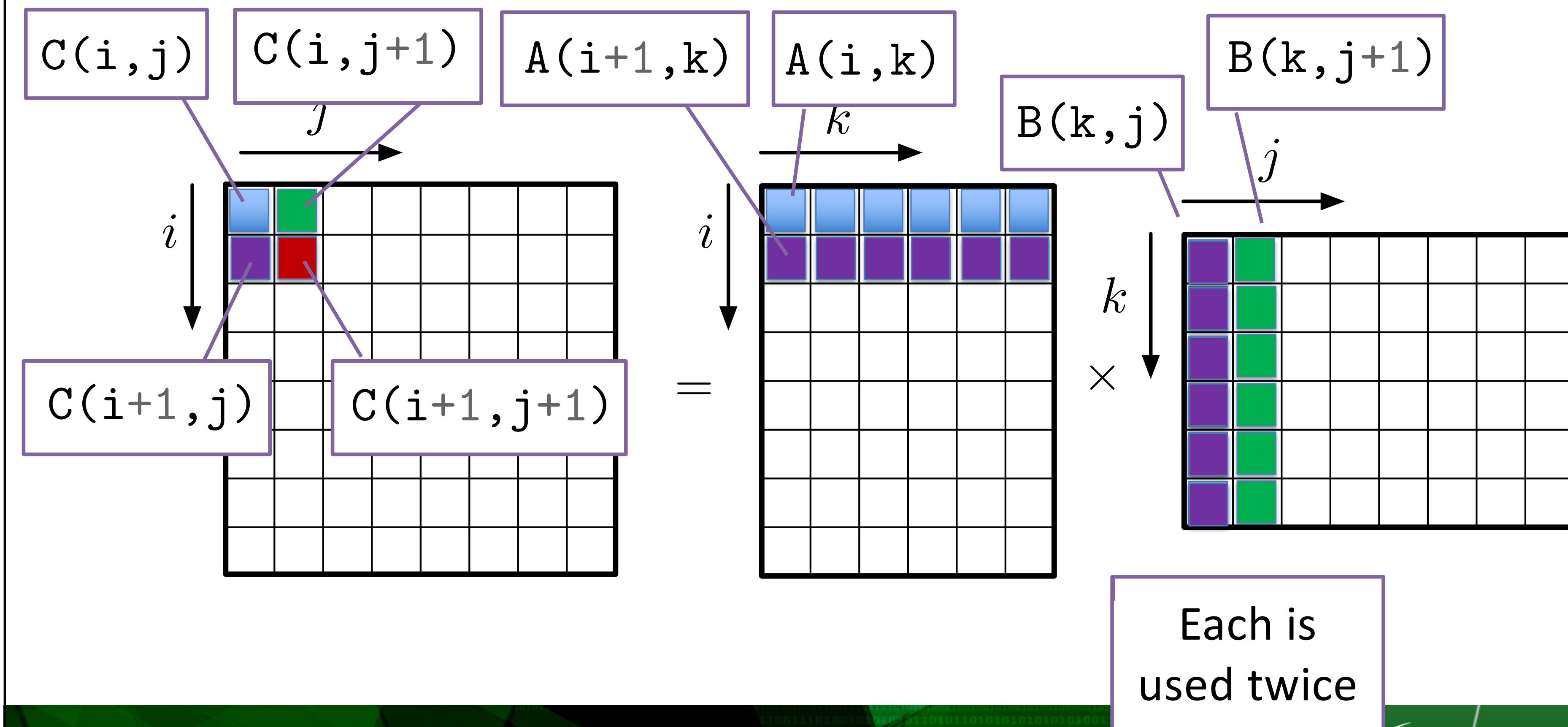
Order of Operations



Order of Operations



Reuse: How Many Times Are Data Reused?



Improving Locality: Unroll and Ja

```
void tiledMultiply2x2(const Matrix& A, const Matrix&B
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
            C(i  , j  ) += A(i  , k) * B(k, j  );
            C(i  , j+1) += A(i  , k) * B(k, j+1);
            C(i+1, j  ) += A(i+1, k) * B(k, j  );
            C(i+1, j+1) += A(i+1, k) * B(k, j+1);
      }
    }
  }
}
```

B(k,j) is used twice

B(k,j+1) is used twice

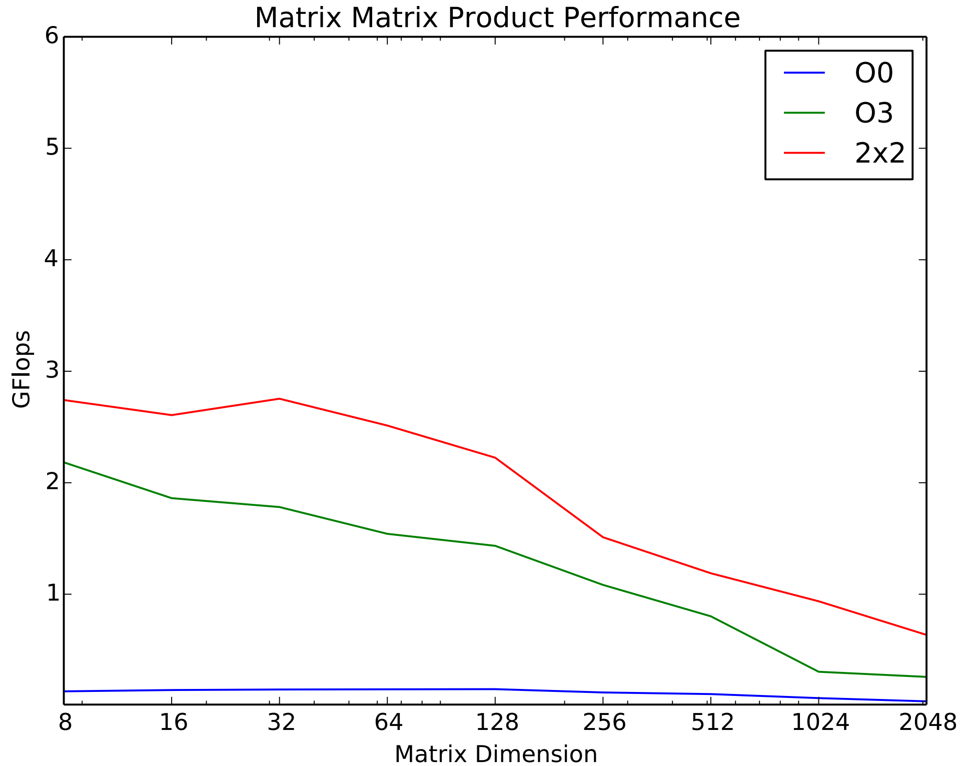
A(i,k) is used twice

Can also hoist (independent of k)

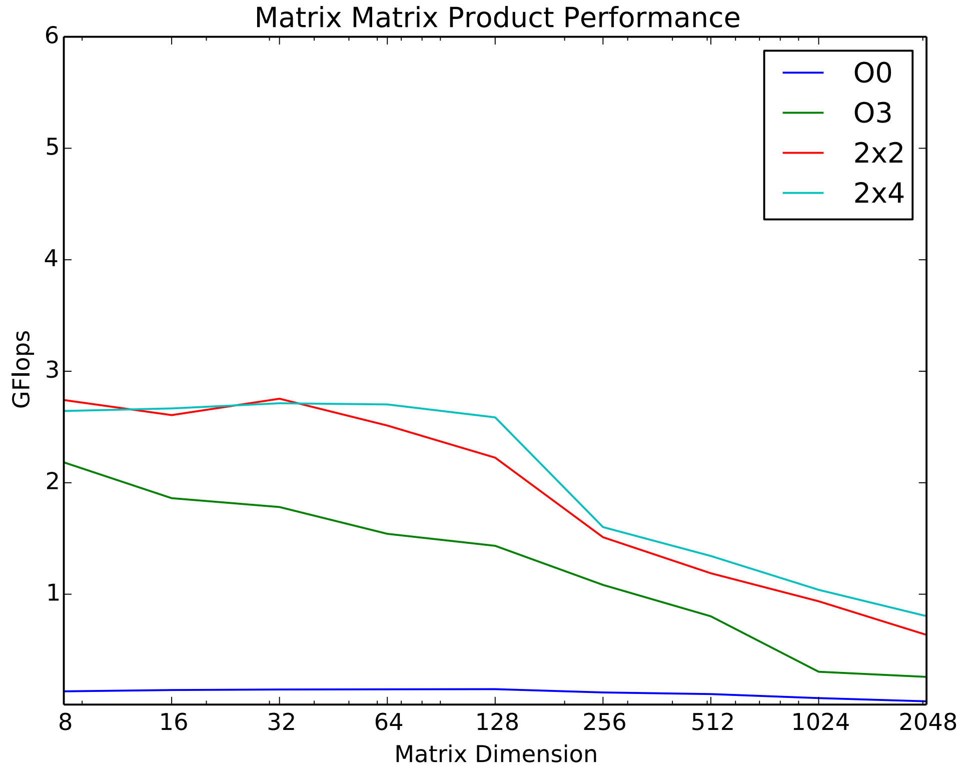
A(i+1,k) is used twice

- Four memory operations and eight floating point operations per iteration
- 8/12 = 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

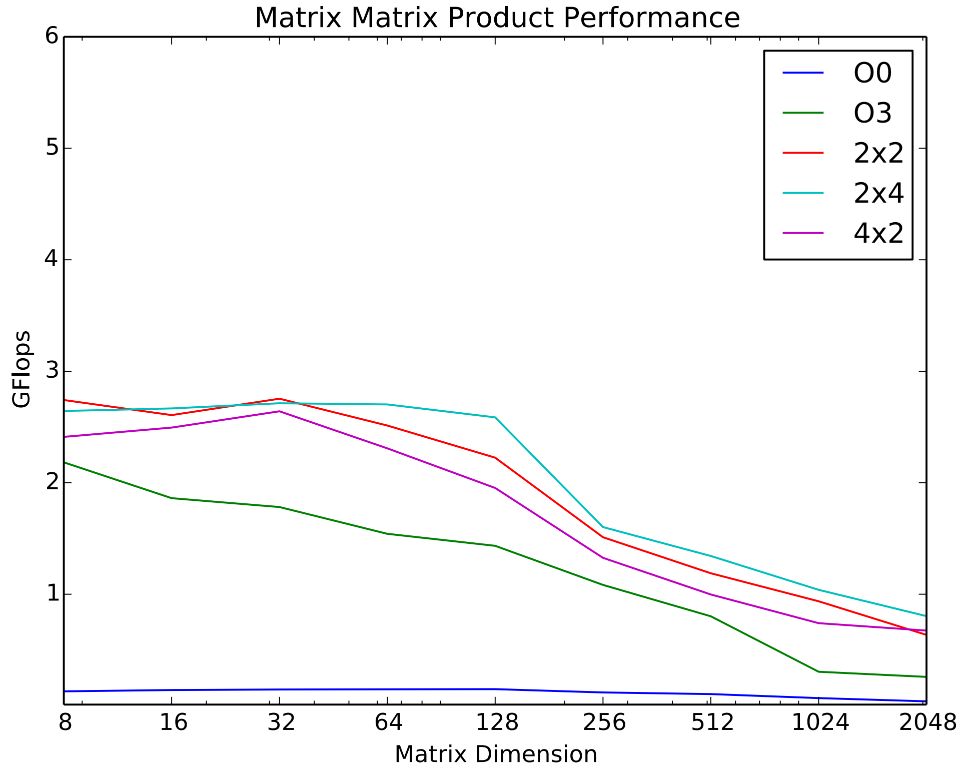
Example: Register Locality



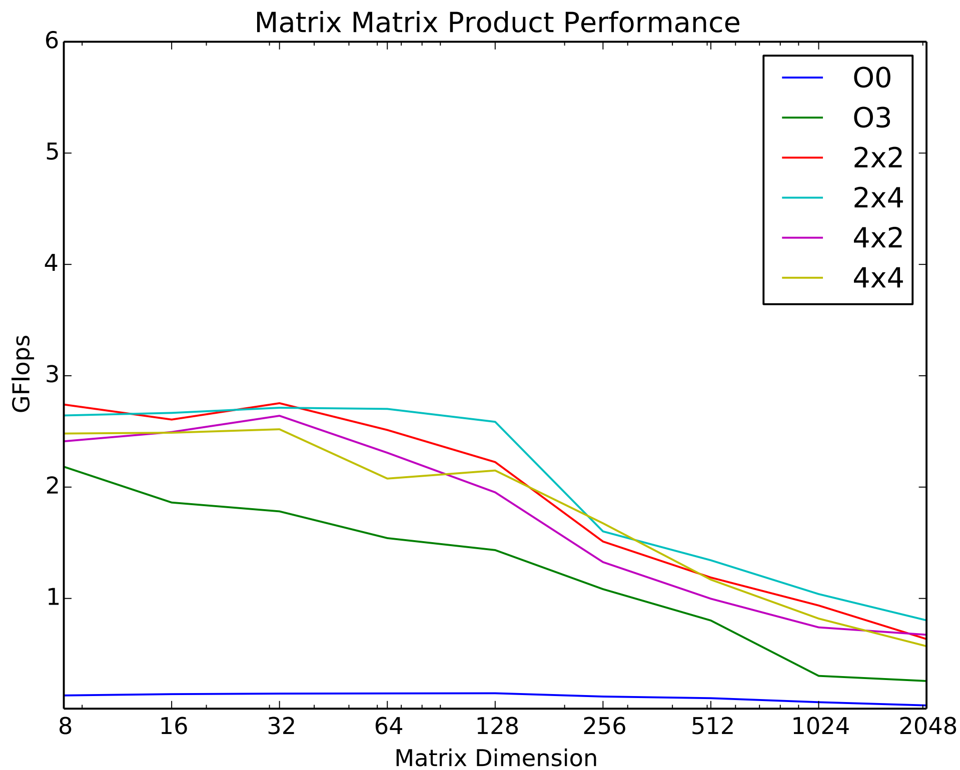
2 by 4


Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4

4 by 2


Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4
4x2

4 by 4



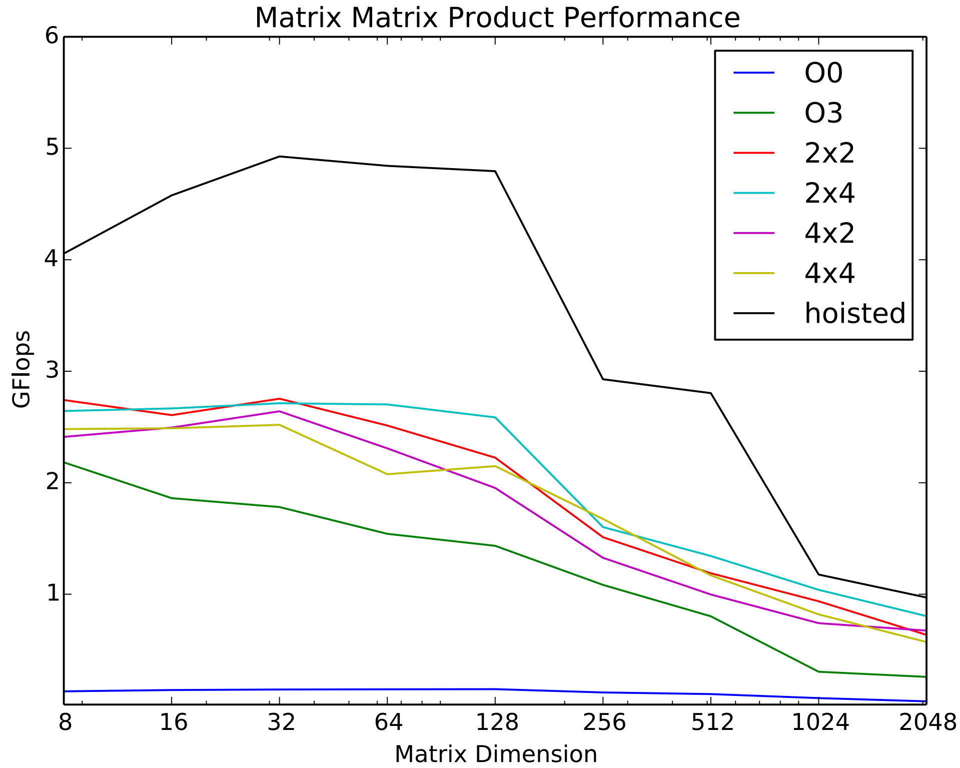
Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4
4x2
4x4

Tiling and Hoisting

```
void hoistedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      double t00 = C(i,  j);     double t01 = C(i,  j+1);
      double t10 = C(i+1,j);     double t11 = C(i+1,j+1);
      for (size_t k = 0; k < A.num_cols(); ++k) {
            t00 += A(i  , k) * B(k, j  );
            t01 += A(i  , k) * B(k, j+1);
            t10 += A(i+1, k) * B(k, j  );
            t11 += A(i+1, k) * B(k, j+1);
      }
      C(i,  j) = t00;  C(i,  j+1) = t01;
      C(i+1,j) = t10;  C(i+1,j+1) = t11;
    }
  }
}
```

Hoist 2x2 tile

Tiling and Hoisting



Hoisting

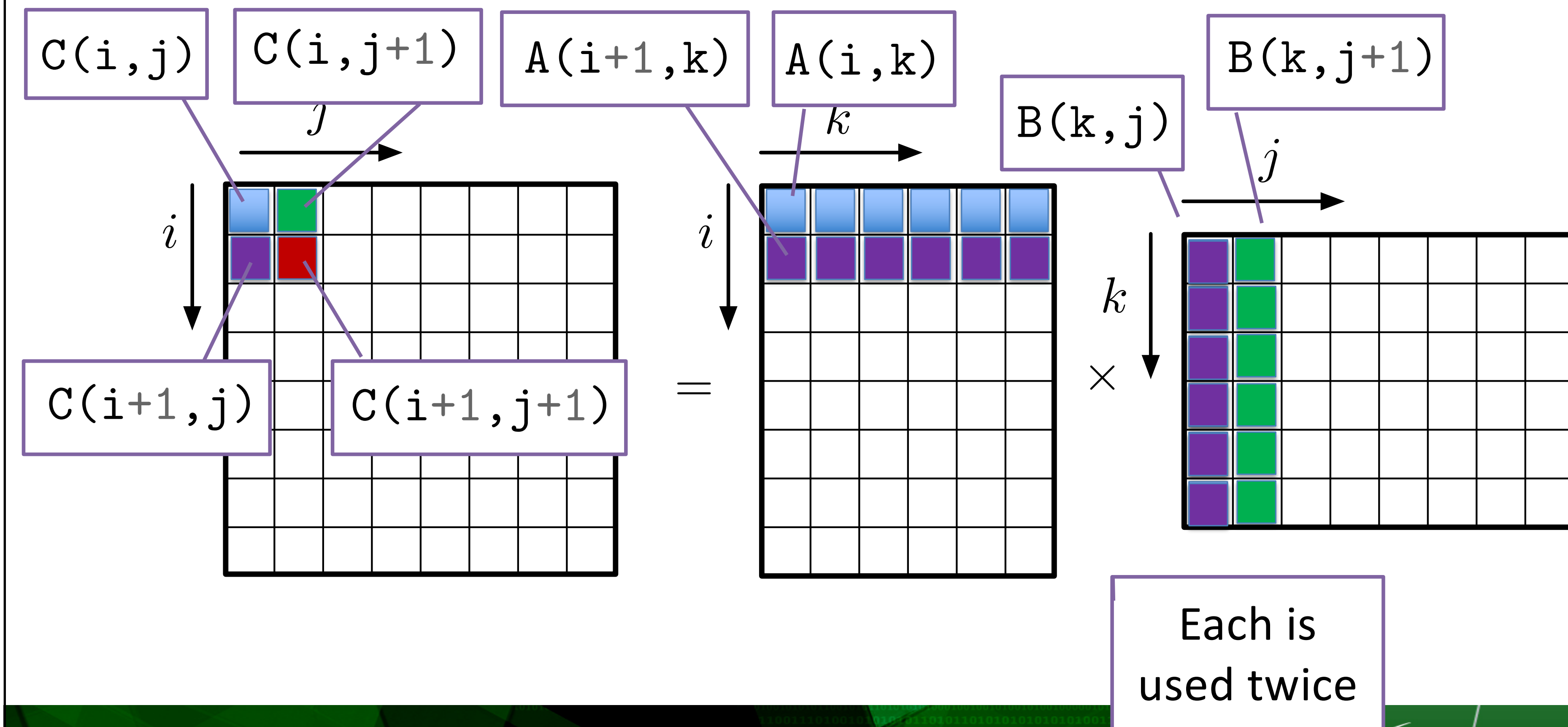
```
void multiply(const Matrix& A, const Matrix& B, Matrix& C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      double t = C(i,j);
      for (size_t k = 0; k < A.num_cols(); ++k) {
        t += A(i,k) * B(k,j);
      }
      C(i,j) = t;
    }
  }
```

Hoist C(i,j)

- Load `A(i,k)` into register
- Load `B(k,j)` into register
- Multiply
- Add

- Two memory operations and two floating point operations per iteration
- 2/4 = 1/2 flop per cycle (if each operation is one cycle)

Reuse: How Many Times Are Data Reused?



Improving Locality: Unroll and J

```
void tiledMultiply2x2(const Matrix& A, const Matrix& B,
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i  , j  ) += A(i  , k) * B(k, j  );
        C(i  , j+1) += A(i  , k) * B(k, j+1);
        C(i+1, j  ) += A(i+1, k) * B(k, j  );
        C(i+1, j+1) += A(i+1, k) * B(k, j+1);
      }
    }
  }
}
```

B(k,j) is used twice

B(k,j+1) is used twice

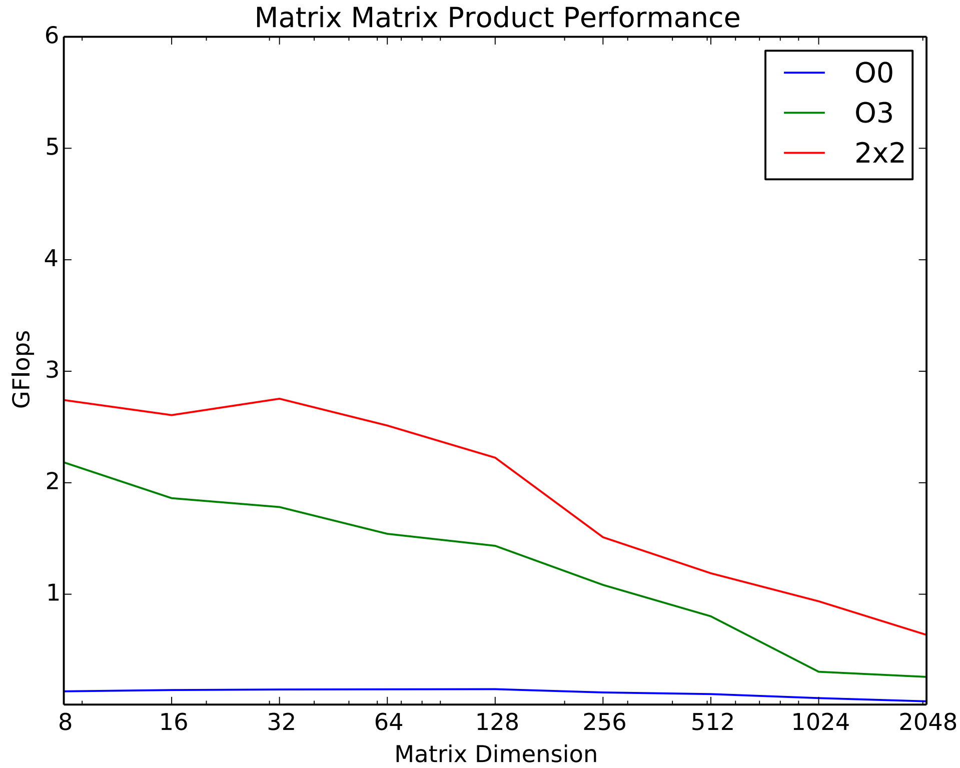
A(i,k) is used twice

A(i+1,k) is used twice

Can also hoist (independent of k)

- Four memory operations and eight floating point operations per iteration
- 8/12 = 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

Example: Register Locality

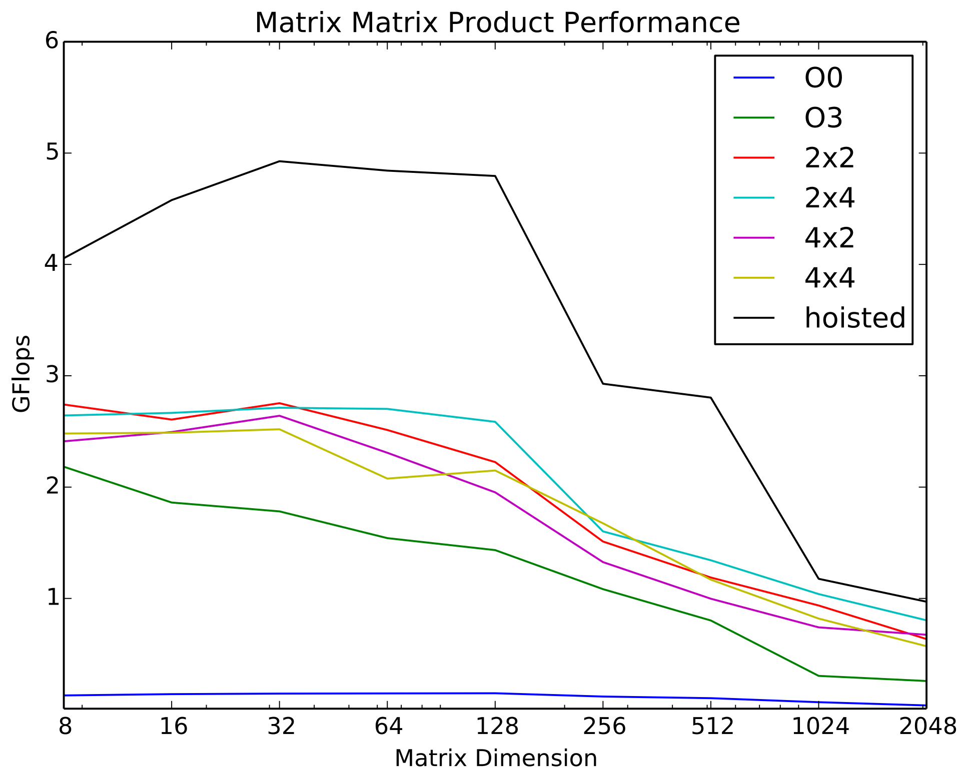


Tiling and Hoisting

```
void hoistedTiledMultiply2x2(const Matrix& A, const Matrix& B, Matrix& C) {
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      double t00 = C(i,j);        double t01 = C(i,j+1);
      double t10 = C(i+1,j);      double t11 = C(i+1,j+1);
      for (size_t k = 0; k < A.num_cols(); ++k) {
        t00 += A(i  , k) * B(k, j  );
        t01 += A(i  , k) * B(k, j+1);
        t10 += A(i+1, k) * B(k, j  );
        t11 += A(i+1, k) * B(k, j+1);
      }
      C(i,  j) = t00;  C(i,  j+1) = t01;
      C(i+1,j) = t10;  C(i+1,j+1) = t11;
    }
  }
}
```

Hoist 2x2 tile

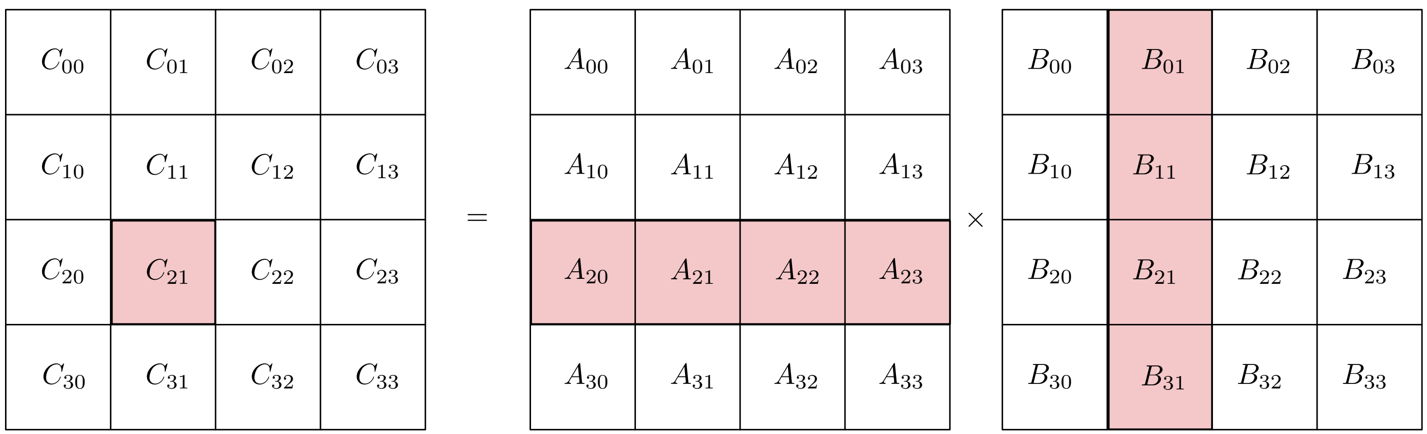
Tiling and Hoisting



Improving Locality: Cache

- Large matrix problems won't fit completely into cache
- Use blocked algorithm – work with blocks that will fit into cache

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$



$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

- Each product term fits completely into cache and runs at high-performance
- Cache misses amortized work with $O(N^3)$ data $O(N^2)$

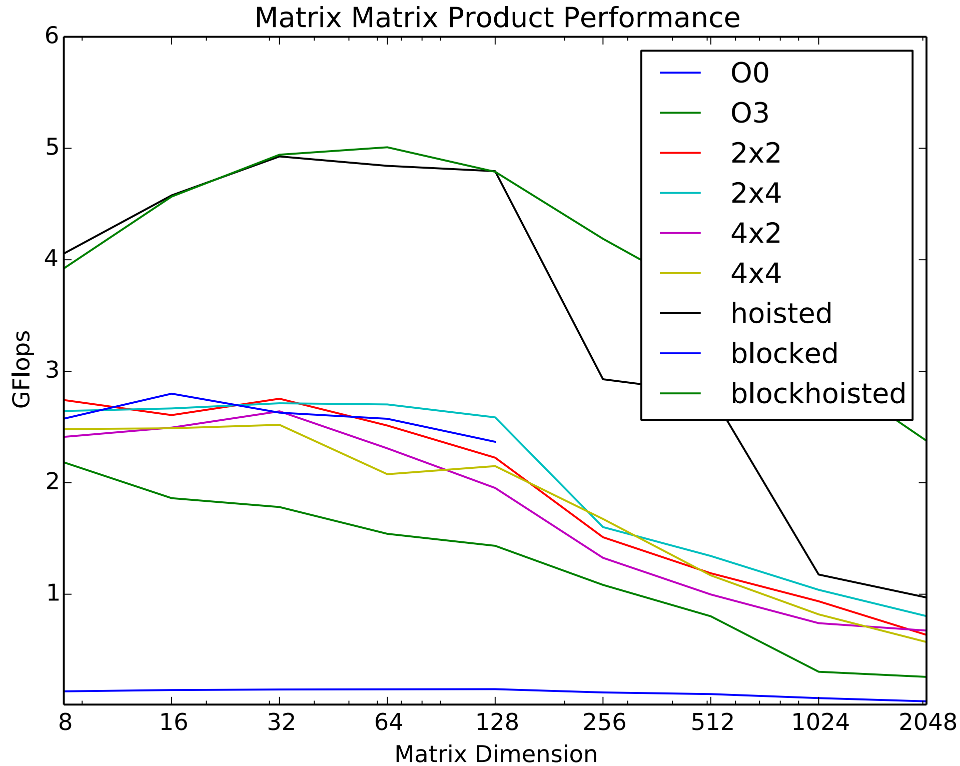
Blocking and Tiling

```
void blockedTiledMultiply2x2(const Matrix& A, const Matrix& B, Matrix& C) {
  const int blocksize = std::min(A.num_rows(), 32);

  for (size_t ii = 0; ii < A.num_rows(); ii += blocksize) {
    for (size_t jj = 0; jj < B.num_cols(); jj += blocksize) {
      for (size_t kk = 0; kk < A.num_cols(); kk += blocksize) {

        for (size_t i = ii; i < ii+blocksize; i += 2) {
          for (size_t j = jj; j < jj+blocksize; j += 2) {
            for (size_t k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
              C(i  , j+1) += A(i  , k) * B(k, j+1);
              C(i+1, j  ) += A(i+1, k) * B(k, j  );
              C(i+1, j+1) += A(i+1, k) * B(k, j+1);
            }
          }
        }
      }
    }
  }
}
```

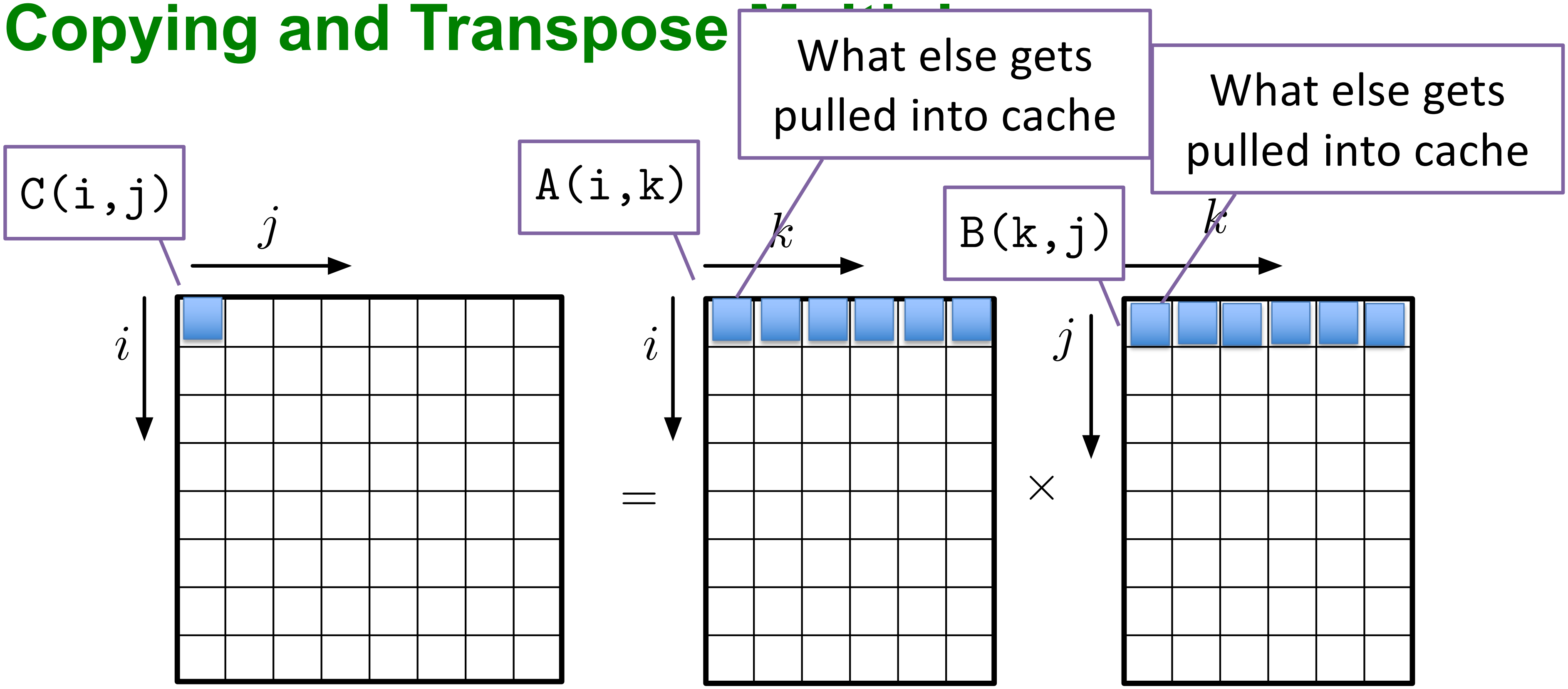
Outer loops work across blocks (for each block)

Inner loops work on blocks

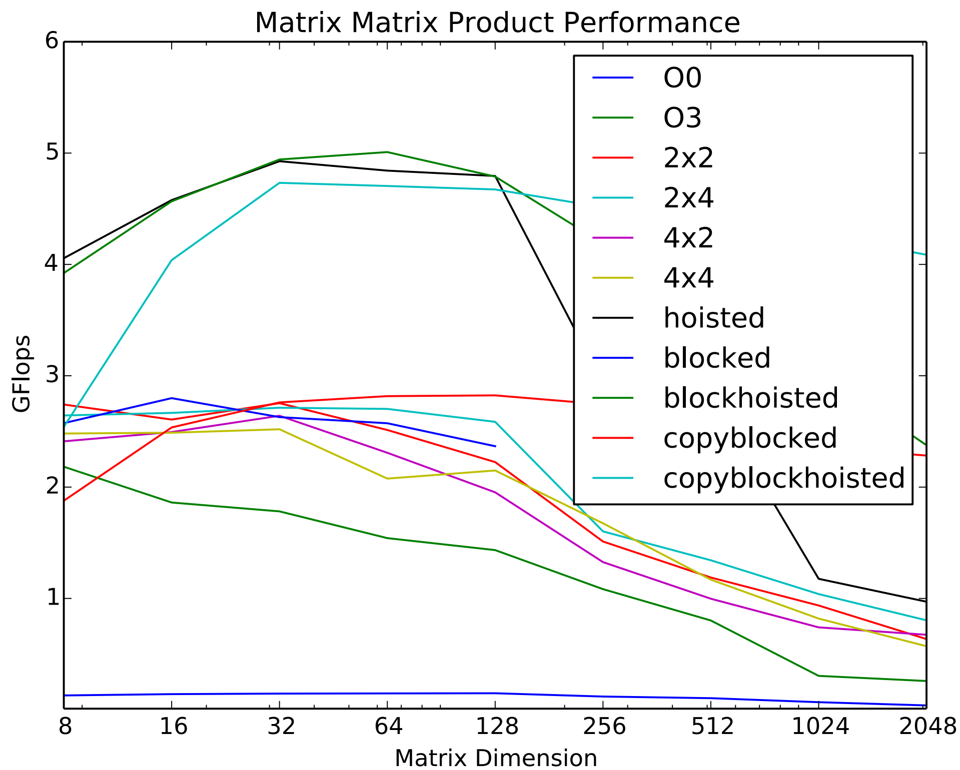
Blocking and Tiling and Hoisting



Copying and Transpose



Blocking and Tiling and Hoisting and Copying



Tuning

- Starting with base code
- Various compiler optimizations help
- Tiling (which size)
- Blocking (what size)
- What size works best for Tiling and Blocking ***together?***

- What loop ordering? Matrix matrix product has six different orderings? What block ordering?

- What about when we add AVX, and threads, etc?

How do we find the optimal combination?

Magic: the power of apparently influencing the course of events by using mysterious or supernatural forces

The answer will be different for different CPUs

Finding the Sweet Spot

- Exhaustive parameter space search
  - Tiling, Blocking, Compiler flags, AVX inst, loop ordering
- Original project at UC Berkeley phiPAC (Bilmes et al)
- Further developed by Whaley and Dongarra → Automatically Tuned Linear Algebra Subprograms (ATLAS)
  - Recently honored with "test of time" award
- (cf) also "Goto" BLAS and FLAME (Goto, van de Geijn)

And wrote a program to generate different multiply functions

This started as a final course project

The competition was to write fastest matrix-matrix product

Students were the good kind of lazy

One Gigantic Bug

What if num_rows is not an integer product of blocksize?

```
void blockedTiledMultiply2x2(const Matrix& A, const Ma
  const int blocksize = std::min(A.numRows(), 32);

  for (int ii = 0; ii < A.numRows(); ii += blocksize)
    for (int jj = 0; jj < B.numCols(); jj += blocksize
      for (int kk = 0; kk < A.numCols(); kk += blocksize) {

        for (int i = ii; i < ii+blocksize; i += 2) {
          for (int j = jj; j < jj+blocksize; j += 2) {
            for (int k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
              C(i  , j+1) += A(i  , k) * B(k, j+1);
              C(i+1, j  ) += A(i+1, k) * B(k, j  );
              C(i+1, j+1) += A(i+1, k) * B(k, j+1);
            }
          }
        }
        for (int i = ii; i < ii+blocksize; ++i) {
          for (int j = jj; j < jj+blocksize; ++j) {
            for (int k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
            }
          }
        }
        for (int i = ii; i < ii+blocksize; ++i) {
          for (int j = jj; j < jj+blocksize; ++j) {
            for (int k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
            }
          }
        }
        for (int i = ii; i < ii+blocksize; ++i) {
          for (int j = jj; j < jj+blocksize; ++j) {
            for (int k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
            }
          }
        }
      }
    }
  }
}
```

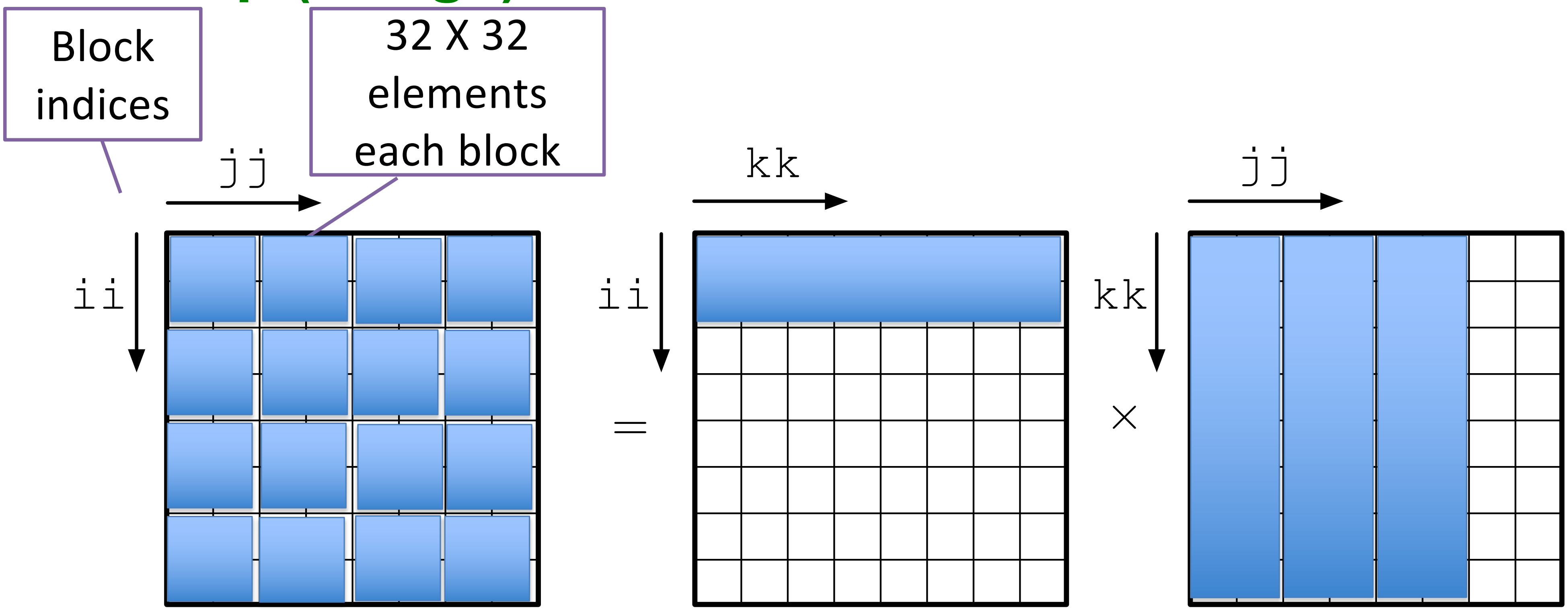
For illustrative purposes only

Example code (slides and source) omit this

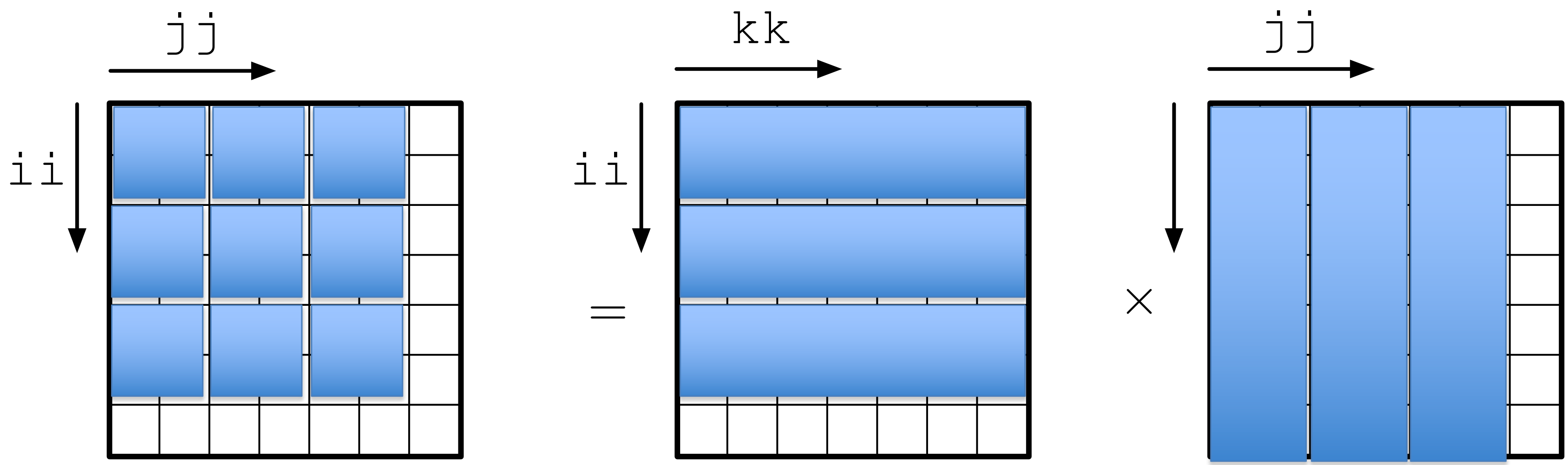
Caveat codor

Powers of 2 are great

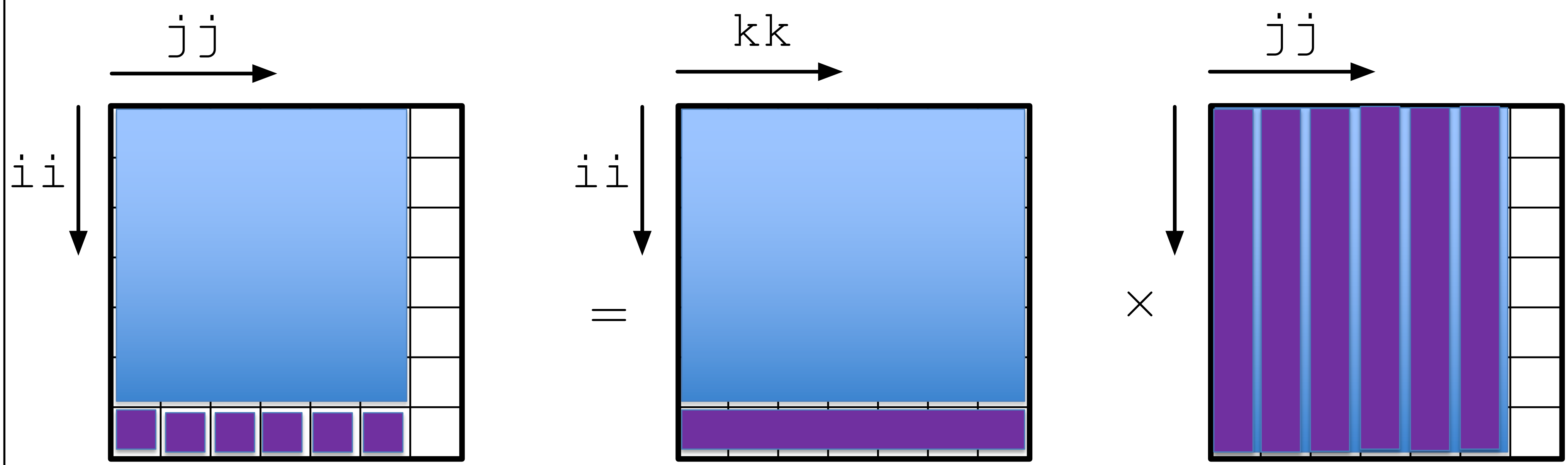
Cleanup (Fringe)



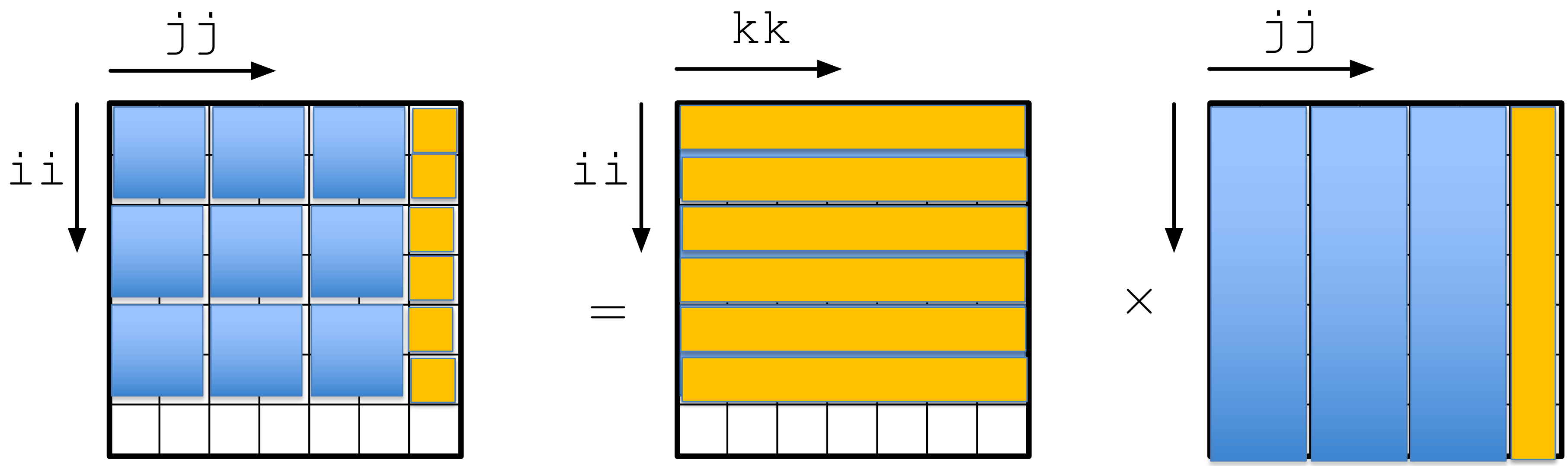
Fringe Cleanup



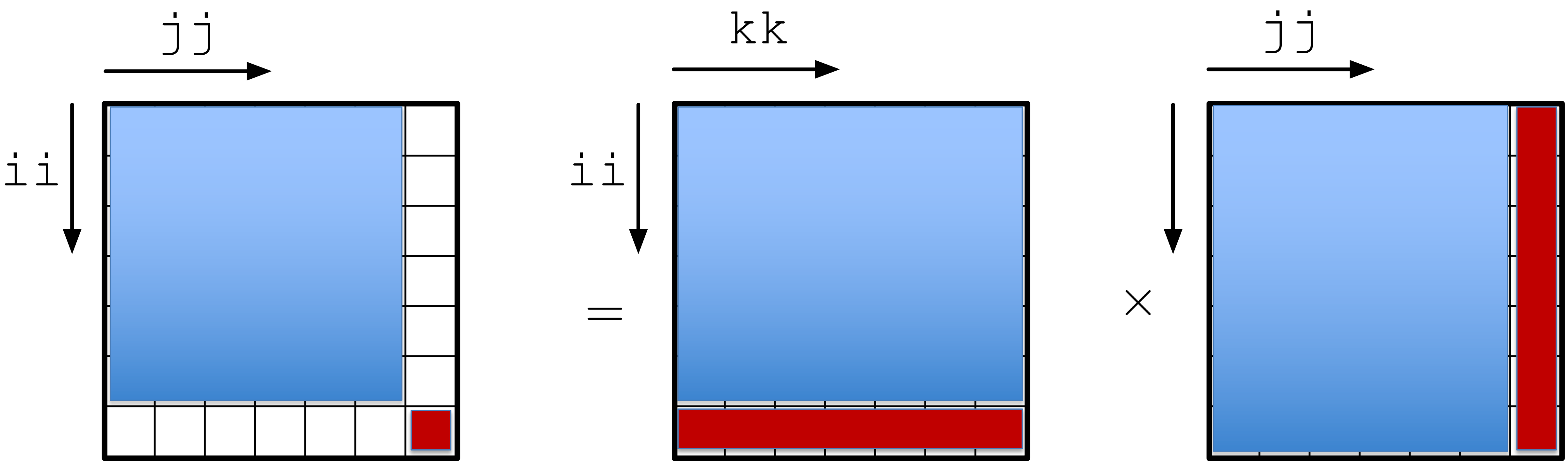
Fringe Cleanup



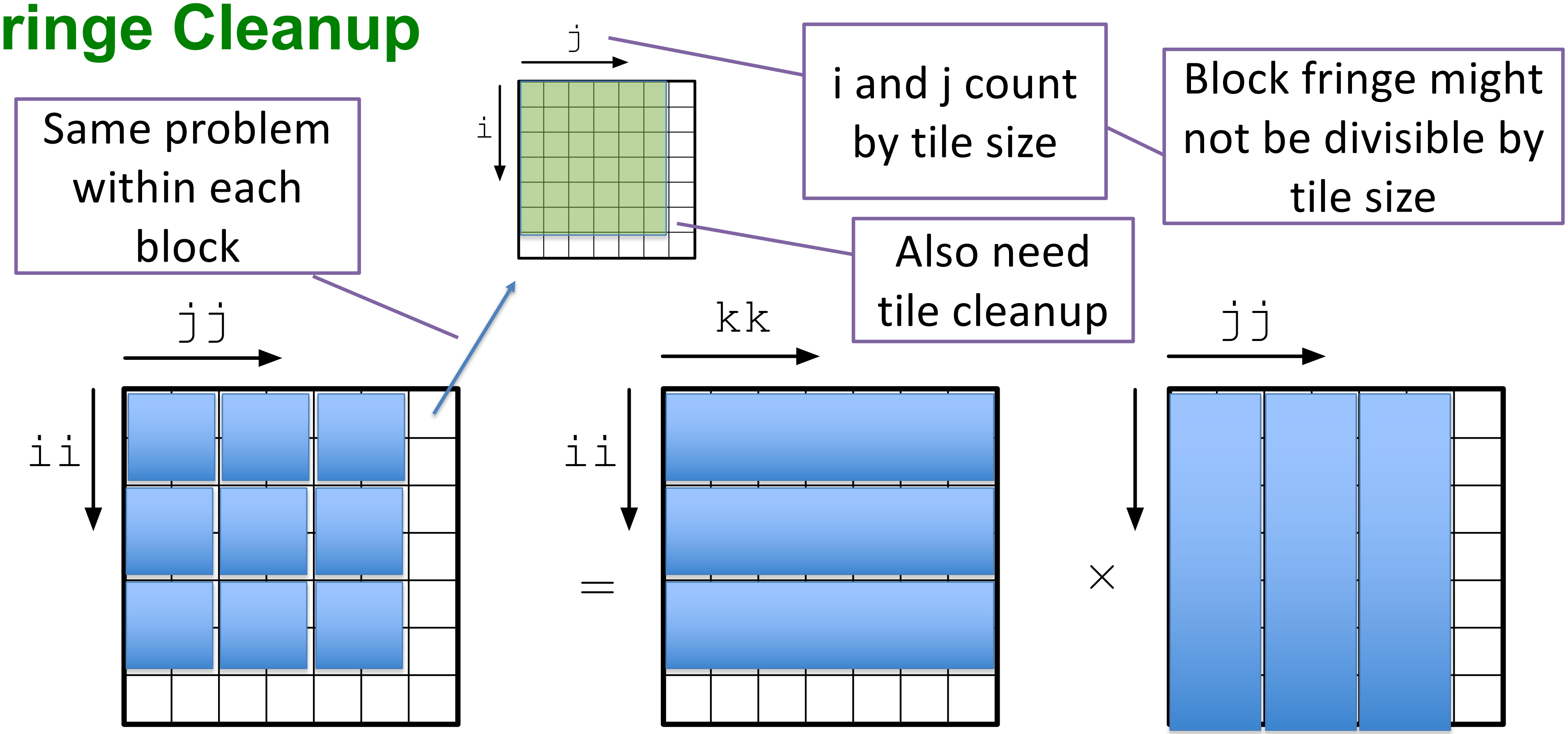
Fringe Cleanup



Fringe Cleanup



Fringe Cleanup

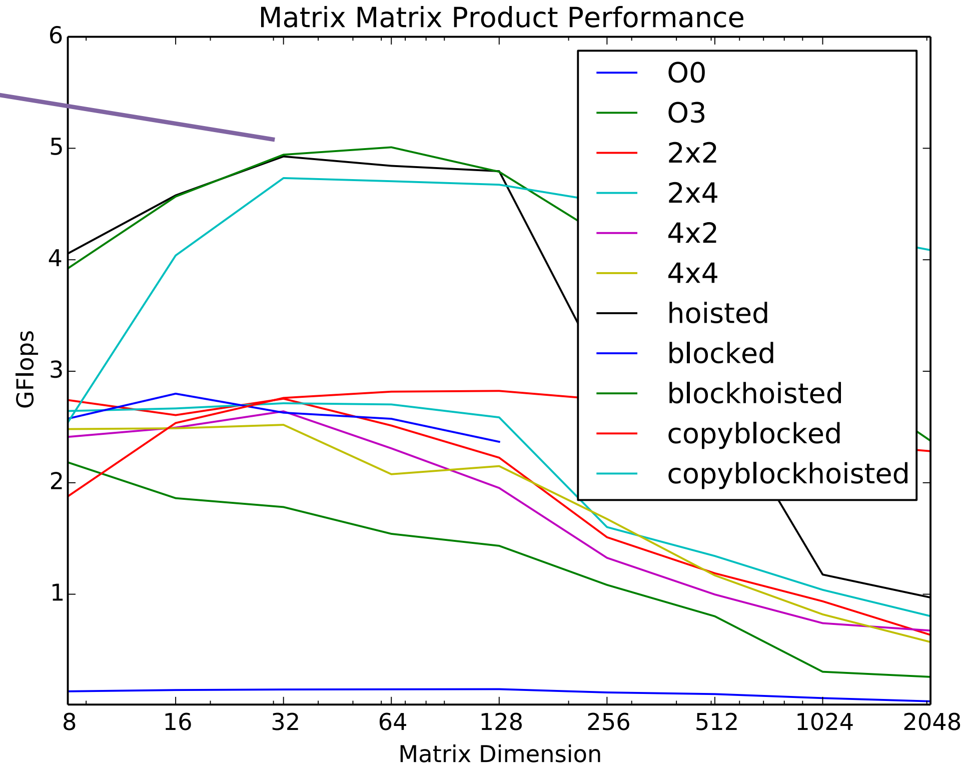


Blocking and Tiling and Hoisting and Copying

Is this the best we can do?

How good is it anyway?

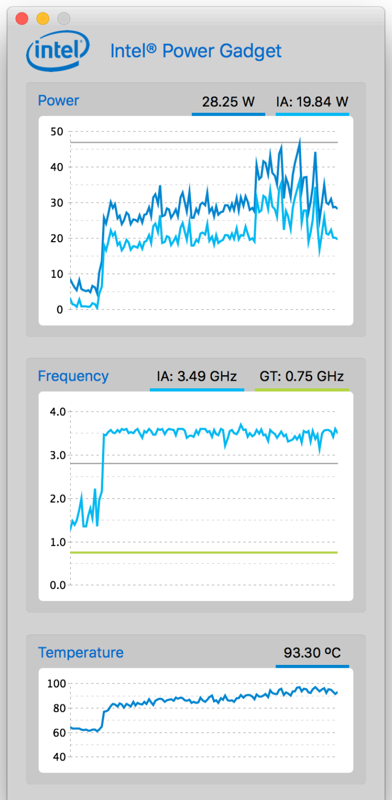
(cf PS 4A)

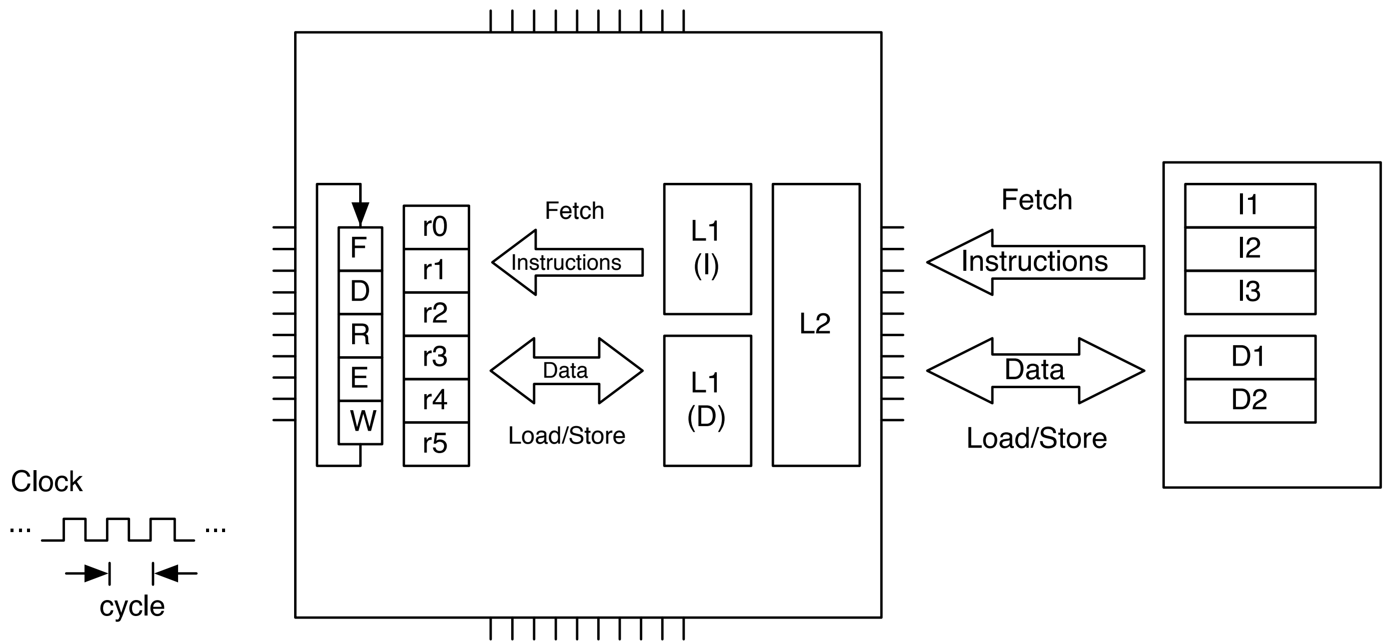


Calypso

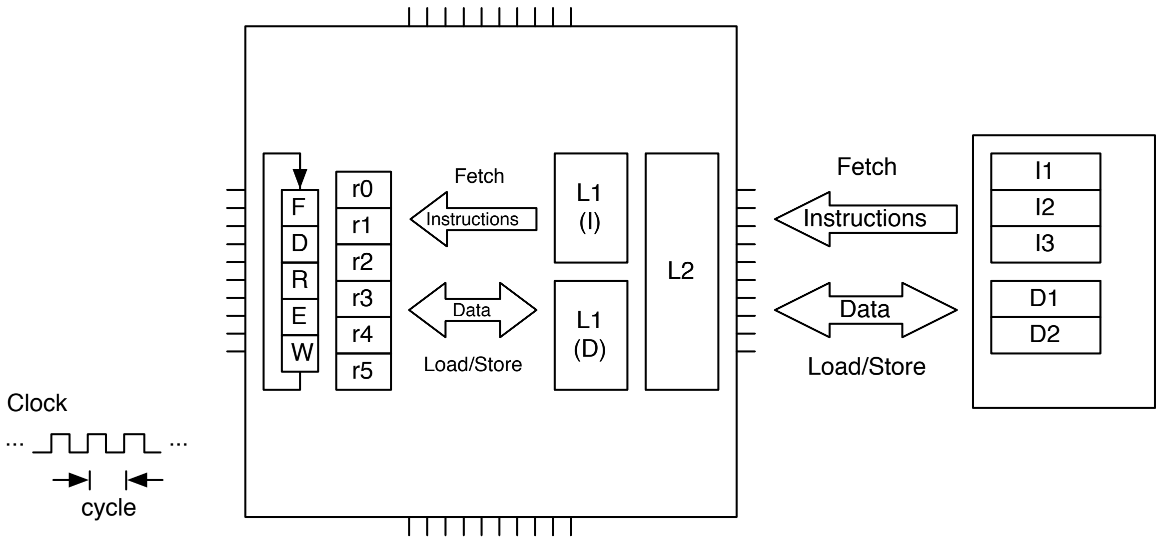
- Program for generating matrix-matrix products

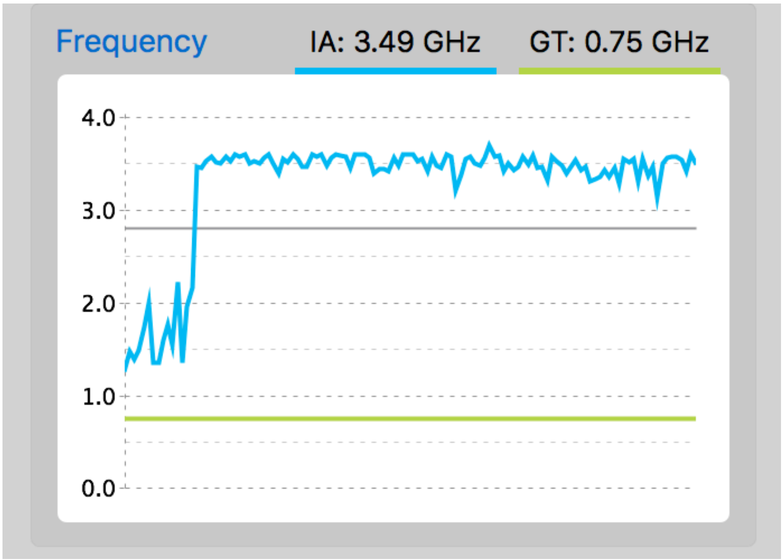
CPU Clock Speed





Peak Performance vs Achieved Performance





$$5 \times 10^9 \frac{\text{FLOPS}}{\text{second}} \div 3.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \approx 1.5 \frac{\text{FLOPS}}{\text{cycle}}$$

Does this make sense?

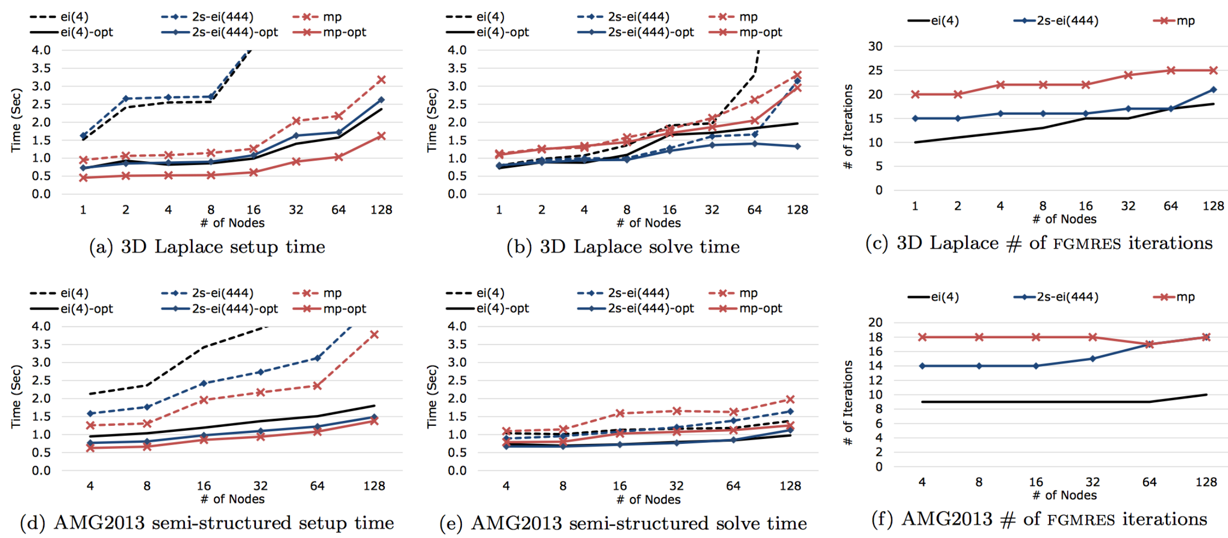
Science

- Frank Harary once suggested the law that **any field that had the word “science” in its name was guaranteed thereby not to be a science.** He would cite as examples Mxxxxx Science, Lxxxxx Science, Pxxxxxx Science, Hxxxxxxxx Science, Sxxxx Science, and **Computer Science**.

But is it Science?

“the pursuit and application of knowledge and understanding of the natural and social world following a systematic methodology based on evidence.”

“Data on how much of the scientific literature is reproducible are rare and generally bleak.”



(a) 3D Laplace setup time

(b) 3D Laplace solve time

(c) 3D Laplace # of FGMRES iterations

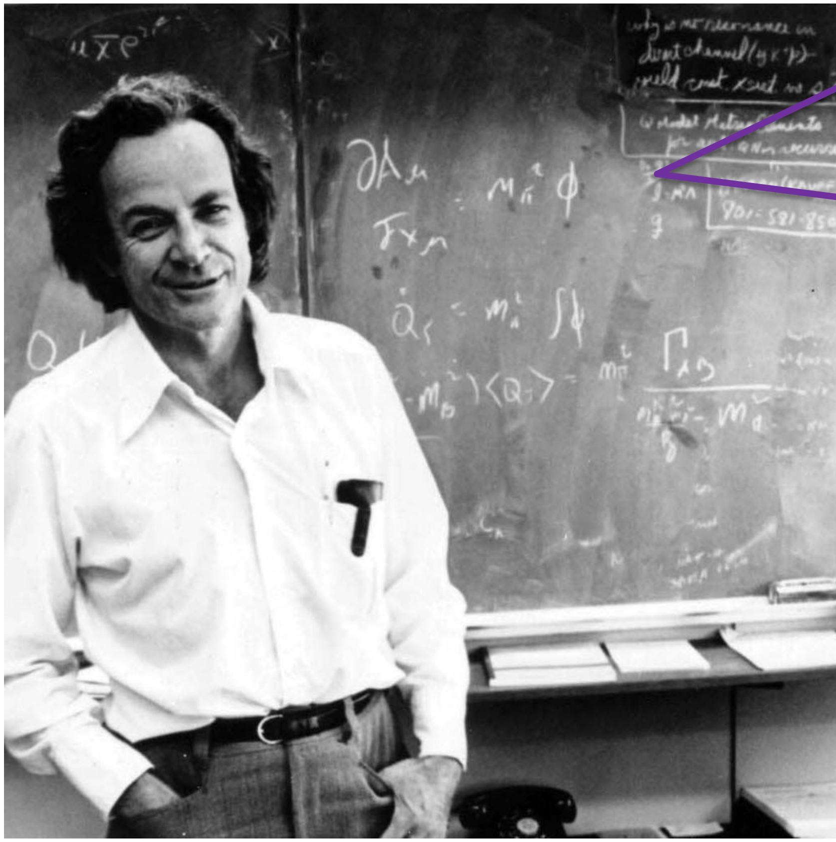
(d) AMG2013 semi-structured setup time

(e) AMG2013 semi-structured solve time

(f) AMG2013 # of FGMRES iterations

Figure 6: Weak scaling multi-node performance (a-c) 3D Laplace matrix with 27-pt discretization from HPCG benchmark [33], ∼27 non-zeros per row, $96^3 \simeq$ 0.9M rows and ∼0.27 GB per rank. (d-e) The semi-structured input from AMG2013 benchmark [35], r=32 and pooldist=1 (generates realistic inputs and requires ≥8 ranks), ∼8 non-zeros per row, ∼1.6M rows and 0.15 GB per rank. The reported times are the maximum among ranks.

Name This Famous Person



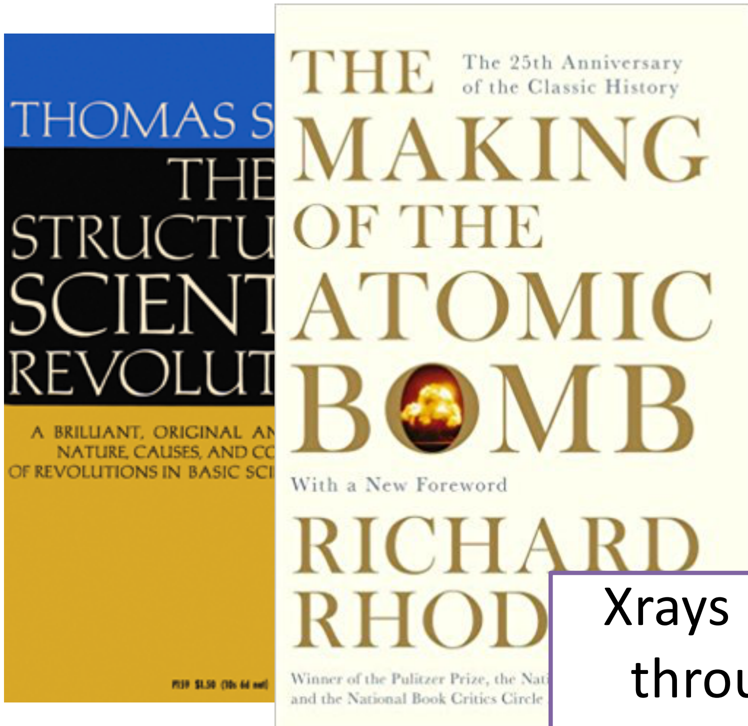
"… [S] scientific integrity, a principle of scientific thought that corresponds to a kind of utter honesty. You must do the best you can—if you know anything at all wrong, or possibly wrong—to explain it.

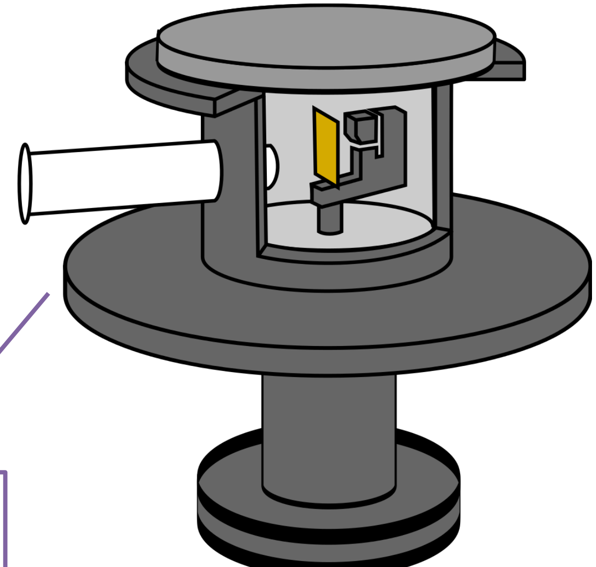
"Cargo Cult Science - Some remarks on science, pseudoscience, and learning how to not fool yourself. Caltech's 1974 commencement address. – Richard Feynman

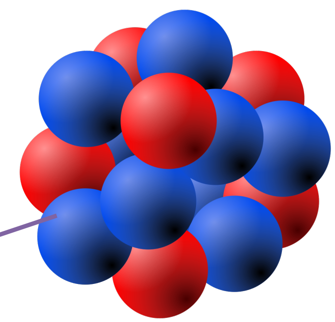
Editorial Comment

- The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" (I found it) but "That's funny"
  - Attributed to Isaac Asimov (and others)

Editorial Comment





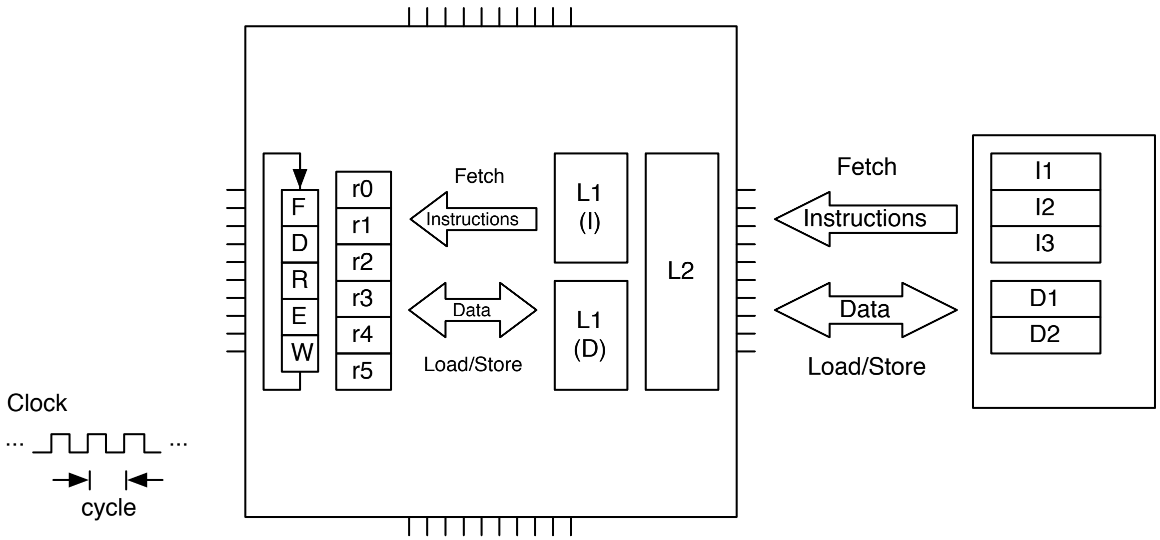


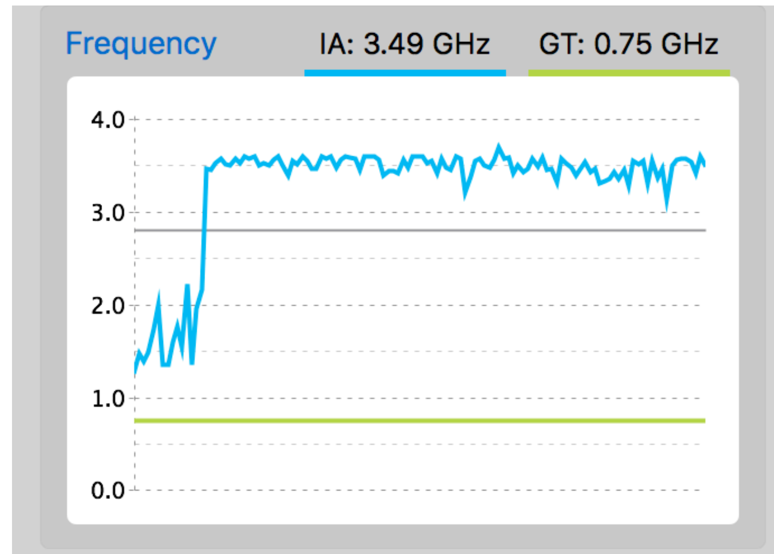
Xrays pass through materials

Alpha particles also, except small bit of noise

Which turned out to be the nucleus

Peak Performance vs Achieved Performance



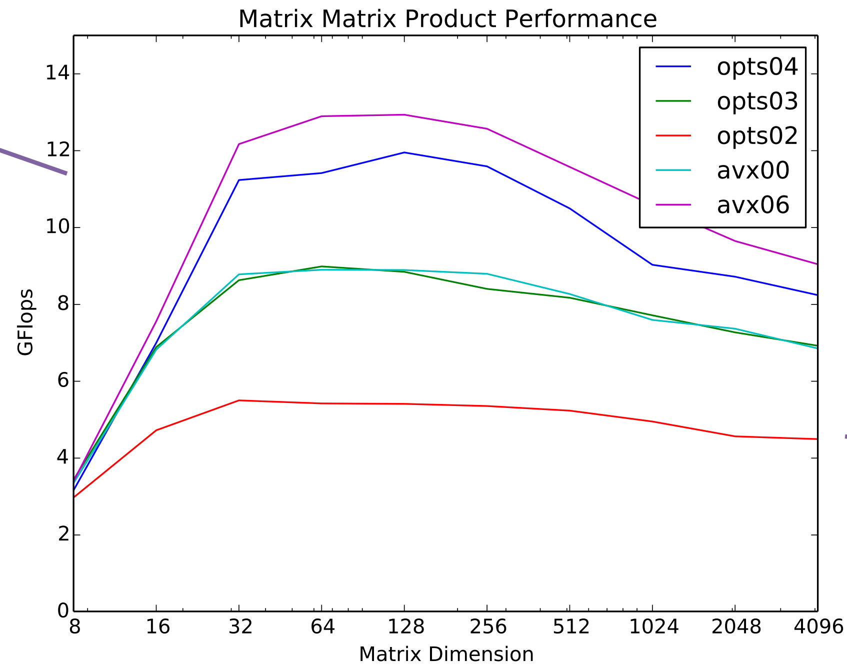


$$5 \times 10^9 \frac{\text{FLOPS}}{\text{second}} \div 3.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \approx 1.5 \frac{\text{FLOPS}}{\text{cycle}}$$

That’s funny

Even Funnier

What magic got these?



Former best performance

$$13 \times 10^9 \frac{\text{FLOPS}}{\text{second}} \div 3.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \approx 3.7 \frac{\text{FLOPS}}{\text{cycle}}$$

Writing Faster Matrix Matrix Product

```
for (int i = ii; i < ii+blocksize; i += 4) {
  for (int j = jj, jb = 0; j < jj+blocksize; j += 4, jb += 4) {

    __m256d t0x = _mm256_load_pd(&C(i,  j));
    __m256d t1x = _mm256_load_pd(&C(i+1,j));
    __m256d t2x = _mm256_load_pd(&C(i+2,j));
    __m256d t3x = _mm256_load_pd(&C(i+3,j));

    for (int k = kk, kb = 0; k < kk+blocksize; ++k, ++kb) {

     __m256d bx = _mm256_setr_pd(BB(jb,kb), BB(jb+1,kb), BB(jb+2,kb), BB(jb+3,kb));

     __m256d a0 = _mm256_broadcast_sd(&A(i   ,k));
      a0  = _mm256_mul_pd(bx,  a0);
      t0x = _mm256_add_pd(t0x, a0);

      __m256d a1 = _mm256_broadcast_sd(&A(i+1,k));
      a1  = _mm256_mul_pd(bx,  a1);
      t1x = _mm256_add_pd(t1x, a1);

      __m256d a2 = _mm256_broadcast_sd(&A(i+2,k));
      a2  = _mm256_mul_pd(bx,  a2);
      t2x = _mm256_add_pd(t2x, a2);

      __m256d a3 = _mm256_broadcast_sd(&A(i+3,k));
      a3  = _mm256_mul_pd(bx,  a3);
      t3x = _mm256_add_pd(t3x, a3);
    }

    _mm256_store_pd(&C(i,  j), t0x);
    _mm256_store_pd(&C(i+1,j), t1x);
    _mm256_store_pd(&C(i+2,j), t2x);
    _mm256_store_pd(&C(i+3,j), t3x);
  }
}
```

Intel advanced vector extensions

(Intrinsics for)

```
__m256d a0 = _mm256_broadcast_sd(&A(i   ,k));
 a0  = _mm256_mul_pd(bx,  a0);
 t0x = _mm256_add_pd(t0x, a0);
```

Vector load

Vector multiply

Vector add

Writing Faster Matrix Matrix Product

```
for (int i = ii; i < ii+blocksize; i += 4) {
  for (int j = jj, jb = 0; j < jj+blocksize; j += 4, jb += 4) {

    __m256d t0x = _mm256_load_pd(&C(i,  j));
    __m256d t1x = _mm256_load_pd(&C(i+1,j));
    __m256d t2x = _mm256_load_pd(&C(i+2,j));
    __m256d t3x = _mm256_load_pd(&C(i+3,j));

    for (int k = kk, kb = 0; k < kk+blocksize; ++k, ++kb) {

     __m256d bx = _mm256_setr_pd(BB(jb,kb), BB(jb+1,kb), BB(jb+2,kb), BB(jb+3,kb));

     __m256d a0 = _mm256_broadcast_sd(&A(i  ,k));
     a0  = _mm256_mul_pd(bx,  a0);
     t0x = _mm256_add_pd(t0x, a0);

     __m256d a1 = _mm256_broadcast_sd(&A(i+1,k));
     a1  = _mm256_mul_pd(bx,  a1);
     t1x = _mm256_add_pd(t1x, a1);

     __m256d a2 = _mm256_broadcast_sd(&A(i+2,k));
     a2  = _mm256_mul_pd(bx,  a2);
     t2x = _mm256_add_pd(t2x, a2);

     __m256d a3 = _mm256_broadcast_sd(&A(i+3,k));
     a3  = _mm256_mul_pd(bx,  a3);
     t3x = _mm256_add_pd(t3x, a3);
    }

    _mm256_store_pd(&C(i,  j), t0x);
    _mm256_store_pd(&C(i+1,j), t1x);
    _mm256_store_pd(&C(i+2,j), t2x);
    _mm256_store_pd(&C(i+3,j), t3x);
  }
}
```

```
__m256d a0 = _mm256_broadcast_sd(&A(i  ,k));
 a0  = _mm256_mul_pd(bx,  a0);
 t0x = _mm256_add_pd(t0x, a0);
```

Double is 8 bytes (64 bits)

256 bit #

256 bit load

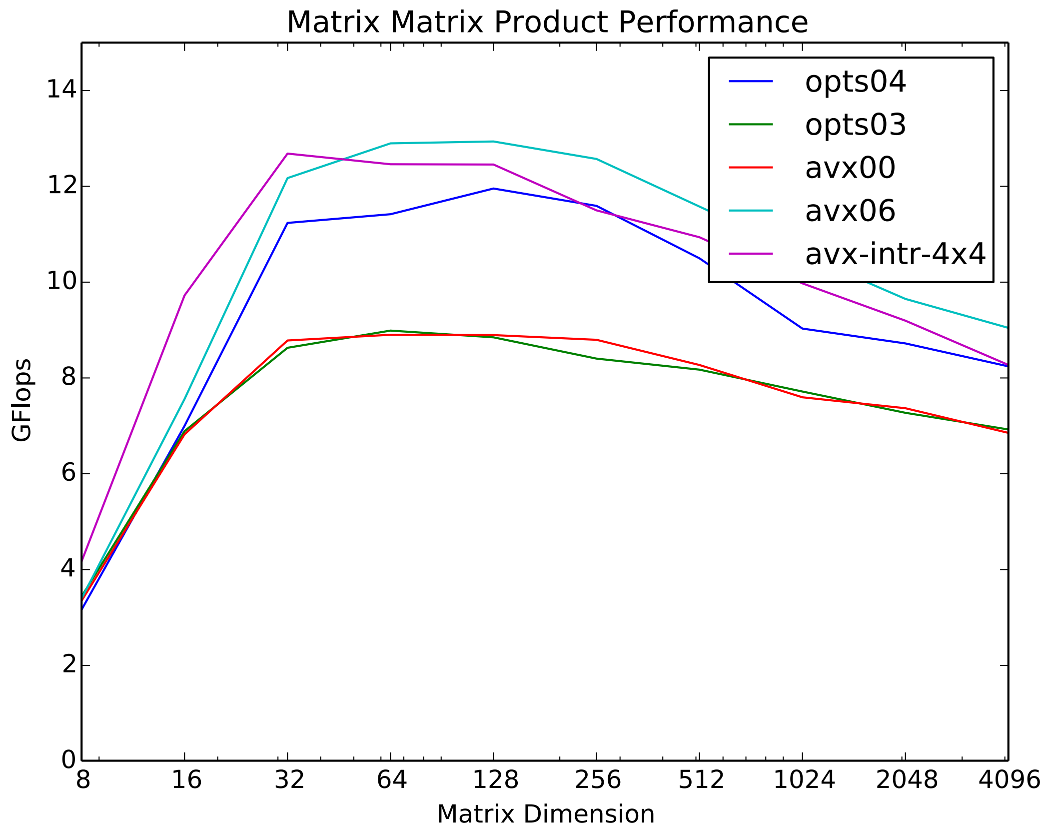
Four doubles

256 bit multiply (1 instruction)

256 bit add

Four FLOPS per cycle

Writing Faster Matrix Matrix Product



Under the Hood

```
for (int i = ii; i < ii+blocksize; i += 4) {
  for (int j = jj, jb = 0; j < jj+blocksize; j += 4, jb += 4) {

    __m256d t0x = _mm256_load_pd(&C(i,  j));
    __m256d t1x = _mm256_load_pd(&C(i+1,j));
    __m256d t2x = _mm256_load_pd(&C(i+2,j));
    __m256d t3x = _mm256_load_pd(&C(i+3,j));

    for (int k = kk, kb = 0; k < kk+blocksize; ++k, ++kb) {

     __m256d bx = _mm256_setr_pd(BB(jb,kb), BB(jb+1,kb), BB(jb+2,kb), BB(jb+3,kb));

     __m256d a0 = _mm256_broadcast_sd(&A(i   ,k));
      a0  = _mm256_mul_pd(bx,  a0);
      t0x = _mm256_add_pd(t0x, a0);

      __m256d a1 = _mm256_broadcast_sd(&A(i+1,k));
      a1  = _mm256_mul_pd(bx,  a1);
      t1x = _mm256_add_pd(t1x, a1);

      __m256d a2 = _mm256_broadcast_sd(&A(i+2,k));
      a2  = _mm256_mul_pd(bx,  a2);
      t2x = _mm256_add_pd(t2x, a2);

      __m256d a3 = _mm256_broadcast_sd(&A(i+3,k));
      a3  = _mm256_mul_pd(bx,  a3);
      t3x = _mm256_add_pd(t3x, a3);
    }

    _mm256_store_pd(&C(i,  j), t0x);
    _mm256_store_pd(&C(i+1,j), t1x);
    _mm256_store_pd(&C(i+2,j), t2x);
    _mm256_store_pd(&C(i+3,j), t3x);
  }
}
```

```
vbroadcastsd     (%rdx,%r8,8), %ymm3
vfmadd213pd      %ymm4, %ymm8, %ymm3
vbroadcastsd     (%rsi,%r8,8), %ymm2
vfmadd213pd      %ymm5, %ymm8, %ymm2
vbroadcastsd     (%rbx,%r8,8), %ymm1
vfmadd213pd      %ymm6, %ymm8, %ymm1
vbroadcastsd     (%rdi,%r8,8), %ymm0
vfmadd213pd      %ymm7, %ymm8, %ymm0
```

X86 Assembly

AVX instructions

256 bit register

Fused Multiply-Add

Multiply-Add are separate here

8 FLOPS per cycle?

Vector Operations from C++

```
for (int i = ii; i < ii+blocksize; i += 2) {
  for (int j = jj, jb = 0; j < jj+blocksize; j += 2, jb += 2) {
    double t00 = C(i,j);       double t01 = C(i,j+1);
    double t10 = C(i+1,j);     double t11 = C(i+1,j+1);

    for (int k = kk, kb = 0; k < kk+blocksize; ++k, ++kb) {
      t00 += A(i  , k) * BB(jb  , kb);
      t01 += A(i  , k) * BB(jb+1, kb);
      t10 += A(i+1, k) * BB(jb  , kb);
      t11 += A(i+1, k) * BB(jb+1, kb);
    }

    C(i,  j) = t00;  C(i,  j+1) = t01;
    C(i+1,j) = t10;  C(i+1,j+1) = t11;
  }
}
```

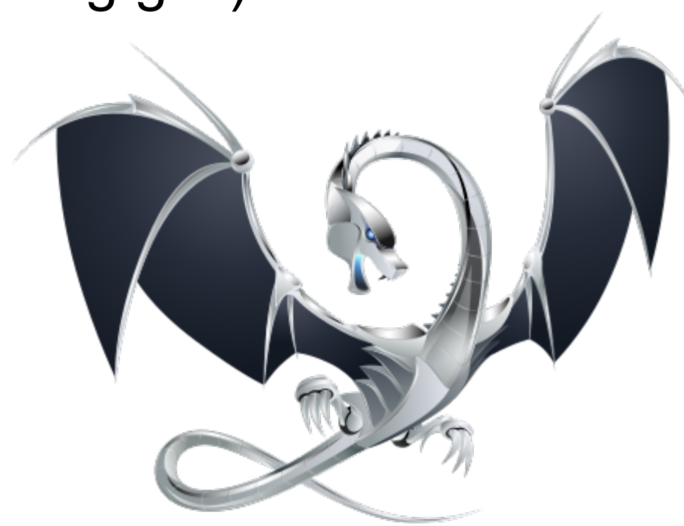
```
vmovupd         (%r8,%r13,8), %ymm4
vmovupd         (%r11,%r13,8), %ymm5
vfmadd231pd     %ymm4, %ymm5, %ymm3
vmovupd -32     (%r9,%r13,8), %ymm6
vfmadd231pd     %ymm4, %ymm6, %ymm2
vmovupd         (%rdx,%r13,8), %ymm4
vfmadd231pd     %ymm5, %ymm4, %ymm1
vfmadd231pd     %ymm6, %ymm4, %ymm0
vmovupd         (%rcx,%r13,8), %ymm4
vmovupd 32      (%r11,%r13,8), %ymm5
vfmadd231pd     %ymm4, %ymm5, %ymm3
vmovupd         (%r9,%r13,8), %ymm6
vfmadd231pd     %ymm4, %ymm6, %ymm2
vmovupd         (%rbx,%r13,8), %ymm4
vfmadd231pd     %ymm5, %ymm4, %ymm1
vfmadd231pd     %ymm6, %ymm4, %ymm0
```

Fused Multiply-Add

256 bit registers

Compilation Process in More Detail (LLVM)

- LLVM (Low Level Virtual Machine) began as a research project at UIUC (Chris Lattner and Vikram Adve)
- Language independent infrastructure for building compilers
- Open source and widely used (supplanting gcc)
- Clang (C-language) front-end
- LLDB debugger

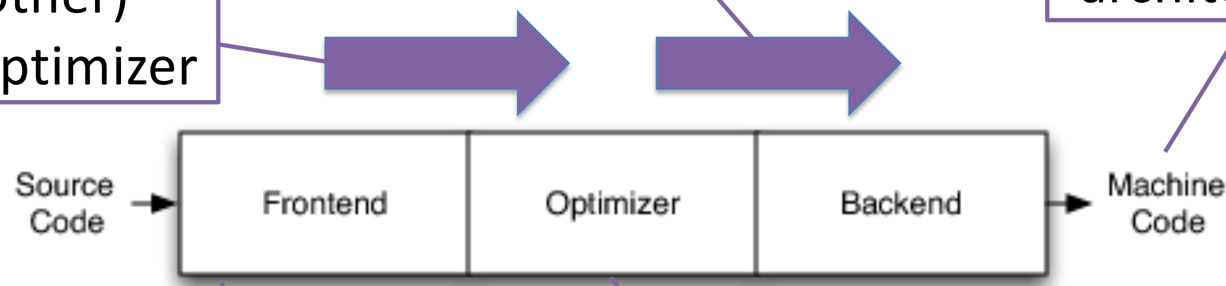


Ideal Compiler

AST (or other) passed to backend

For specific architecture target

AST (or other) passed to optimizer



Generates abstract syntax tree (AST)

Language and architecture independent

How much effort for M languages and N architectures?

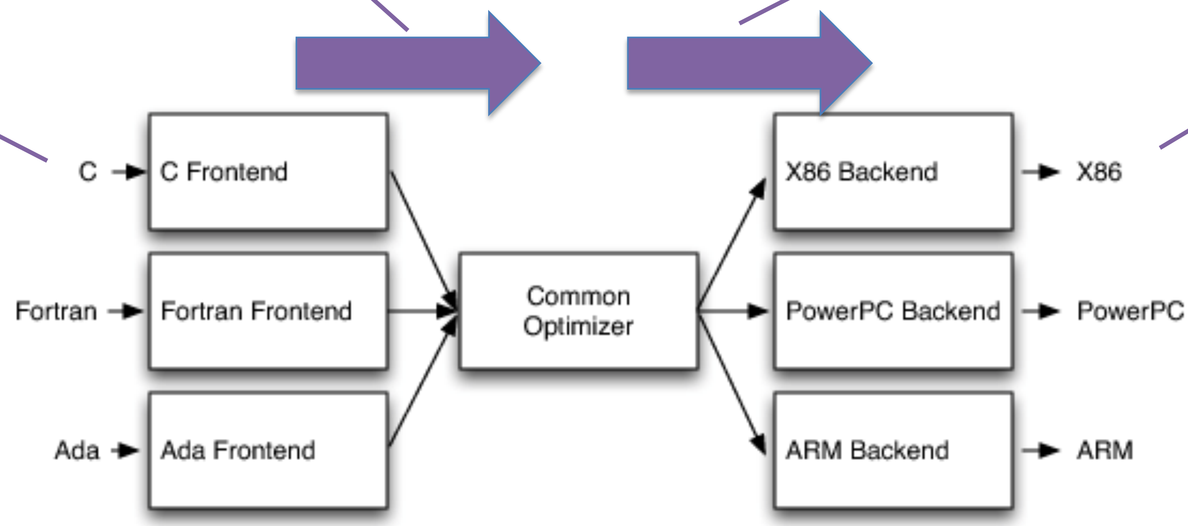
Ideal Compiler

AST (or other) passed to optimizer

AST (or other) passed to backend

Multiple languages

Multiple architectures



How much effort for M languages and N architectures?

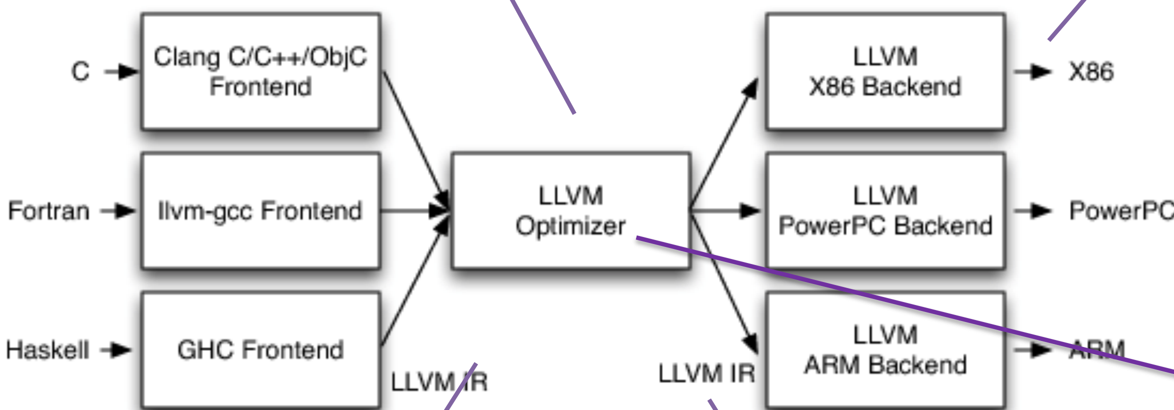
(In theory)

LLVM

Core of LLVM

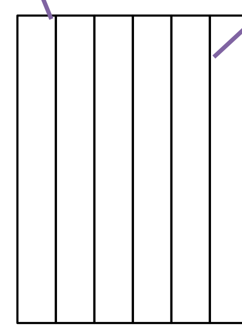
Highly configurable

Highly configurable



Multiple passes

Also with well defined interfaces



LLVM IR is universal compiler language

Well defined interface: in and out

Compiler Optimizations

```
$ echo 'int;' | $(CXX) -xc++ $(CXXFLAGS) - -o /dev/null -\#\#\#
```

- Many options
- -Ofast
- -march=native

```
lums658@WE31821=> make optreport
echo 'int;' | c++ -xc -Ofast -march=native -DNDEBUG -fslp-vectorize-aggressive -mxsave -mavx -mavx2  -std=c++14 -Wc++14-extensions -fslp-vectorize-aggressive -mxsave -mavx -mavx2 -Wall - -o /dev/null -\#\#\#
Apple LLVM version 8.1.0 (clang-802.0.41)
Target: x86_64-apple-darwin16.5.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang" "-cc1" "-triple" "x86_64-apple-macosx10.12.0" "-Wdeprecated-objc-isa-usage" "-Werror=deprecated-objc-isa-usage" "-emit-obj" "-disable-free" "-disable-llvm-verifier" "-discard-value-names" "-main-file-name" "-" "-mrelocation-model" "pic" "-pic-level" "2" "-mthread-model" "posix" "-mdisable-fp-elim" "-menable-no-infs" "-menable-no-nans" "-menable-unsafe-fp-math" "-fno-signed-zeros" "-freciprocal-math" "-ffp-contract=fast" "-ffast-math" "-masm-verbose" "-munwind-tables" "-target-cpu" "haswell" "-target-feature" "+sse2" "-target-feature" "+cx16" "-target-feature" "-tbm" "-target-feature" "-avx512ifma" "-target-feature" "-avx512dq" "-target-feature" "-fma4" "-target-feature" "-prfchw" "-target-feature" "+bmi2" "-target-feature" "-xsavec" "-target-feature" "+fsgsbase" "-target-feature" "+popcnt" "-target-feature" "+aes" "-target-feature" "-pcommit" "-target-feature" "-xsaves" "-target-feature" "-avx512er" "-target-feature" "-clwb" "-target-feature" "-avx512f" "-target-feature" "-pku" "-target-feature" "-smap" "-target-feature" "+mmx" "-target-feature" "-xop" "-target-feature" "-rdseed" "-target-feature" "-hle" "-target-feature" "-sse4a" "-target-feature" "-avx512bw" "-target-feature" "-clflushopt" "-target-feature" "-avx512vl" "-target-feature" "+invpcid" "-target-feature" "-avx512cd" "-target-feature" "-rtm" "-target-feature" "+fma" "-target-feature" "+bmi" "-target-feature" "-mwaitx" "-target-feature" "+rdrnd" "-target-feature" "+sse4.1" "-target-feature" "+sse4.2" "-target-feature" "+sse" "-target-feature" "+lzcnt" "-target-feature" "+pclmul" "-target-feature" "-prefetchwt1" "-target-feature" "+f16c" "-target-feature" "+ssse3" "-target-feature" "-sgx" "-target-feature" "+cmov" "-target-feature" "-avx512vbmi" "-target-feature" "+movbe" "-target-feature" "+xsaveopt" "-target-feature" "-sha" "-target-feature" "-adx" "-target-feature" "-avx512pf" "-target-feature" "+sse3" "-target-feature" "+xsave" "-target-feature" "+avx" "-target-feature" "+avx2" "-target-linker-version" "278.4" "-dwarf-column-info" "-debugger-tuning=lldb" "-resource-dir" "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/8.1.0" "-D" "NDEBUG" "-Ofast" "-Wc++14-extensions" "-Wall" "-std=c++14" "-fdebug-compilation-dir" "/Users/lums658/git/amath-583/src" "-ferror-limit" "19" "-fmessage-length" "96" "-stack-protector" "1" "-fblocks" "-fobjc-runtime=macosx-10.12.0" "-fencode-extended-block-signature" "-fmax-type-align=16" "-fdiagnostics-show-option" "-fcolor-diagnostics" "-vectorize-loops" "-vectorize-slp" "-vectorize-slp-aggressive" "-o" "/var/folders/4z/vn0681g52rx8b18_q2r1fcv0lzfm0s/T/--7075ee.o" "-x" "c" "-"
 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/ld" "-demangle" "-lto_library" "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/libLTO.dylib" "-dynamic" "-arch" "x86_64" "-macosx_version_min" "10.12.0" "-o" "/dev/null" "/var/folders/4z/vn0681g52rx8b18_q2r1fcv0lzfm0s/T/--7075ee.o" "-lc++" "-lSystem" "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/8.1.0/lib/darwin/libclang_rt.osx.a"
lums658@WE31821=>
```

Compiler Diagnostics

- There are some flags to see what the compiler is doing

```
optflags         :
        echo 'int;' | $(CXX) -xc++ $(CXXFLAGS) - -o /dev/null -\#\#\#

defreport        :
        $(CXX) -dM -E -x c++ /dev/null

Matrix.o .       :
        $(CXX) -c $(CXXFLAGS) -Rpass=.* -o Matrix.o
```

Print flags passed to compiler

Print internal #defines

Print what optimizations are applied (and where)

Internal #define

defreport :

```
$(CXX) -dM -E -x c++ /dev/null
```

```
#define OBJC_NEW_PROPERTIES 1
#define _LP64 1
#define __APPLE_CC__ 6000
#define __APPLE__ 1
#define __ATOMIC_ACQUIRE 2
#define __ATOMIC_ACQ_REL 4
#define __ATOMIC_CONSUME 1
#define __ATOMIC_RELAXED 0
#define __ATOMIC_RELEASE 3
#define __ATOMIC_SEQ_CST 5
#define __BLOCKS__ 1
#define __CHAR16_TYPE__ unsigned short
#define __CHAR32_TYPE__ unsigned int
```

340+ total

Very useful for conditional compilation

```
#ifdef __AVX__
    __m128d a = _mm256_extractf128_pd(tx, 0);
    __m128d b = _mm256_extractf128_pd(tx, 1);
    _mm_store_pd(&C(i,j)    , a);
    _mm_store_pd(&C(i+1, j), b);
#endif // __AVX__
```

Optimization Report

```
Matrix.o          :
                  $(CXX) -c $(CXXFLAGS) -Rpass=.* -o Matrix.o
Matrix.cpp: 52: 7: remark: vectorized loop (vectorization width: 4, interleaved count: 4) [-
      for (int k = 0; k < A.numCols(); ++k) {
      ^
Matrix.cpp: 52: 7: remark: unrolled loop by a factor of 2 with run-time trip count [-Rpass=]
Matrix.cpp: 50: 5: remark: unrolled loop by a factor of 8 with run-time trip count [-Rpass=]
    for (int j = 0; j < B.numCols(); ++j) {
```

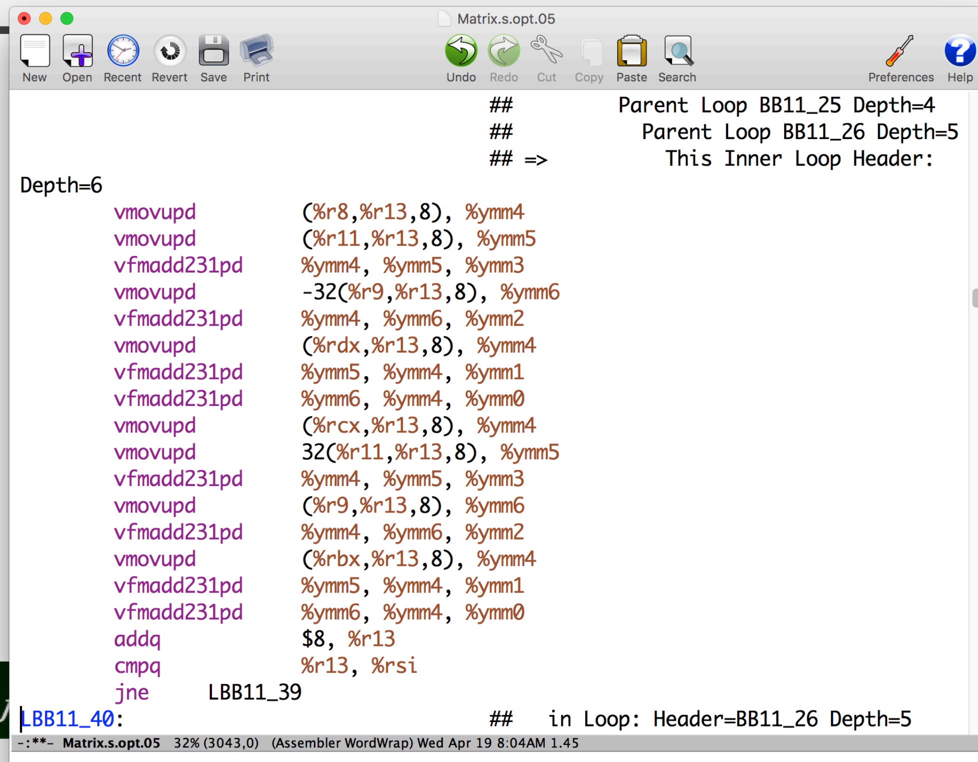
```
    for (int j = 0; j < B.numCols(); ++j) {
      double t = C(i,j);
      for (int k = 0; k < A.numCols(); ++k) {
        t += A(i,k) * B(k,j);
      }
      C(i,j) = t;
    }
```

Selects all

Unroll
Vectorization
Inline

As a Last Resort

```
%.s          : %.cpp
             $(CXX) -S $(CXXFLAGS) $<
```



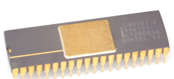
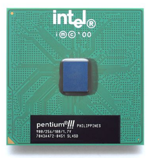
Advanced Vector Extensions

SIMD?

Multi Media Extensions

Streaming SIMD Extensions

Advanced Vector Extensions

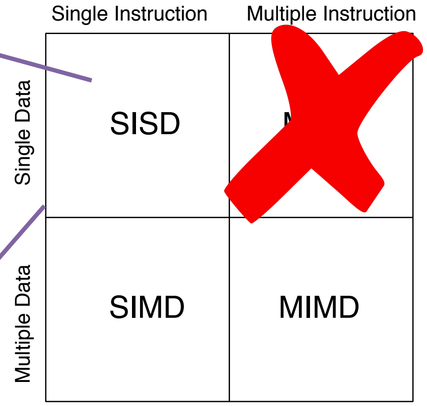
1980	8087		8 80-bit stack registers
1997	MMX		8 64-bit registers
1999	SSE		8 128-bit registers (single) (xmm)
2001	SSE2		8 128-bit registers (double)
2004 2006 2007 2008	SSE3 SSE3.1 SSE4.1 SSE4.2		Many new instructions
2011	AVX		16 256-bit registers (ymm)
2013	AVX2		
2016	AVX512		32 512-bit registers (zmm)

Flynn's Taxonomy (Aside)

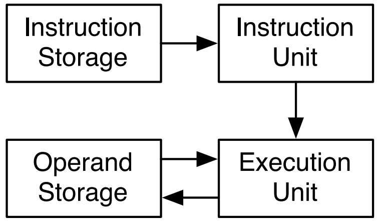
Anyone in HPC must know Flynn's taxonomy

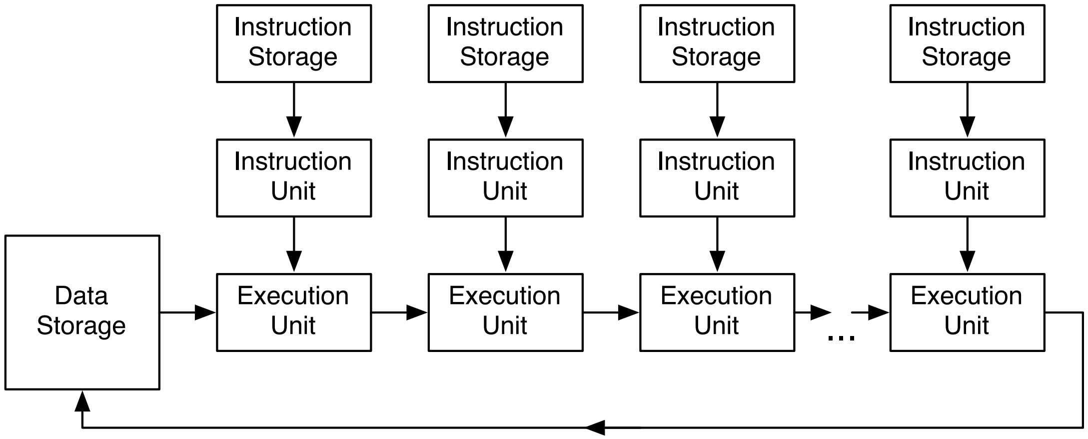
- **Classic** classification of parallel architectures (Michael Flynn, 1966)

Plain old sequential



Based on multiplicity of instruction streams, data storage





SIMD and MIMD

- Two principal parallel computing paradigms (multiple c

But each have their own data

Single instruction at a time

All execution units execute in (c)lock step

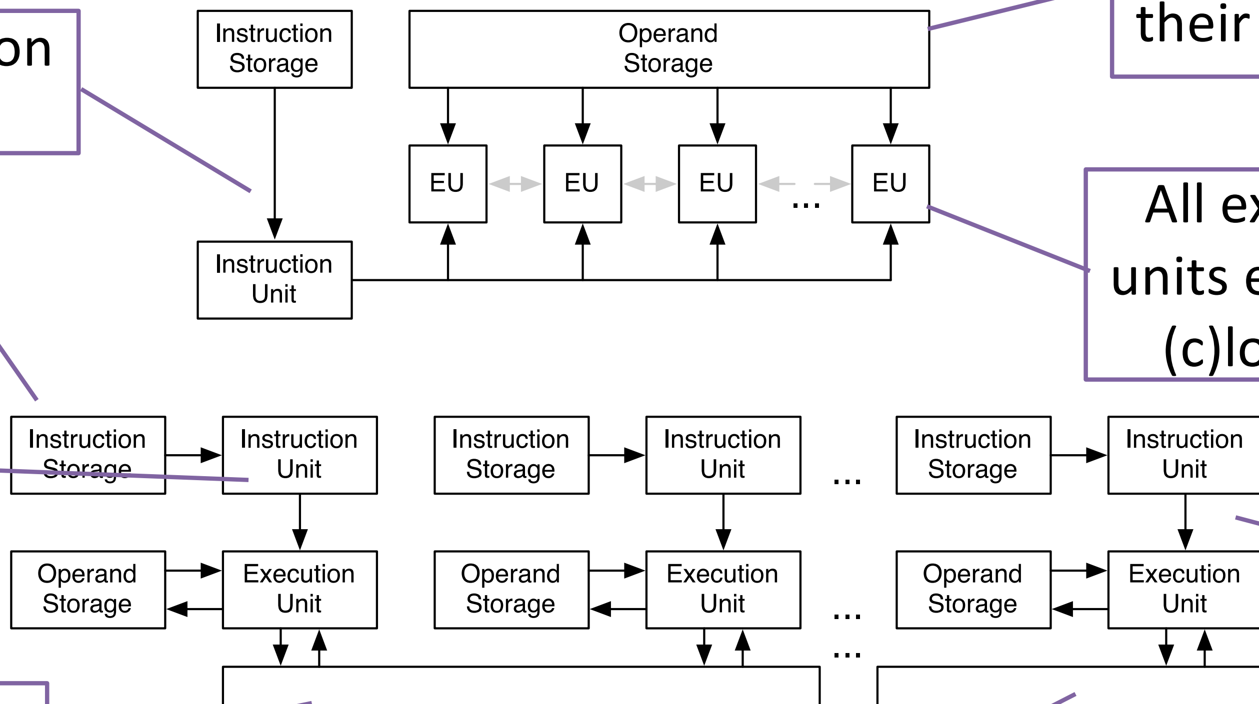
Multiple instruction

EUs run independently (w own instrs)

Coming up next

Shared Memory

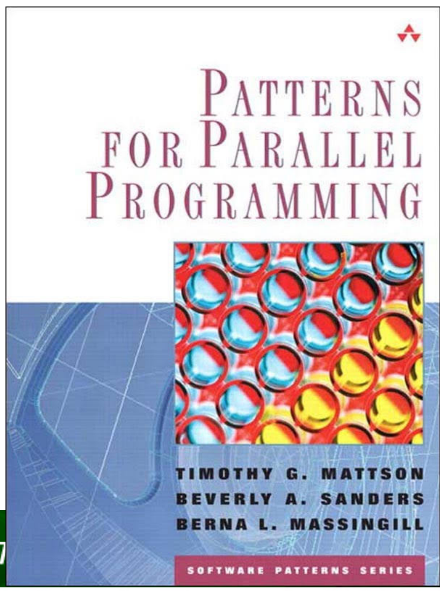
Not Shared

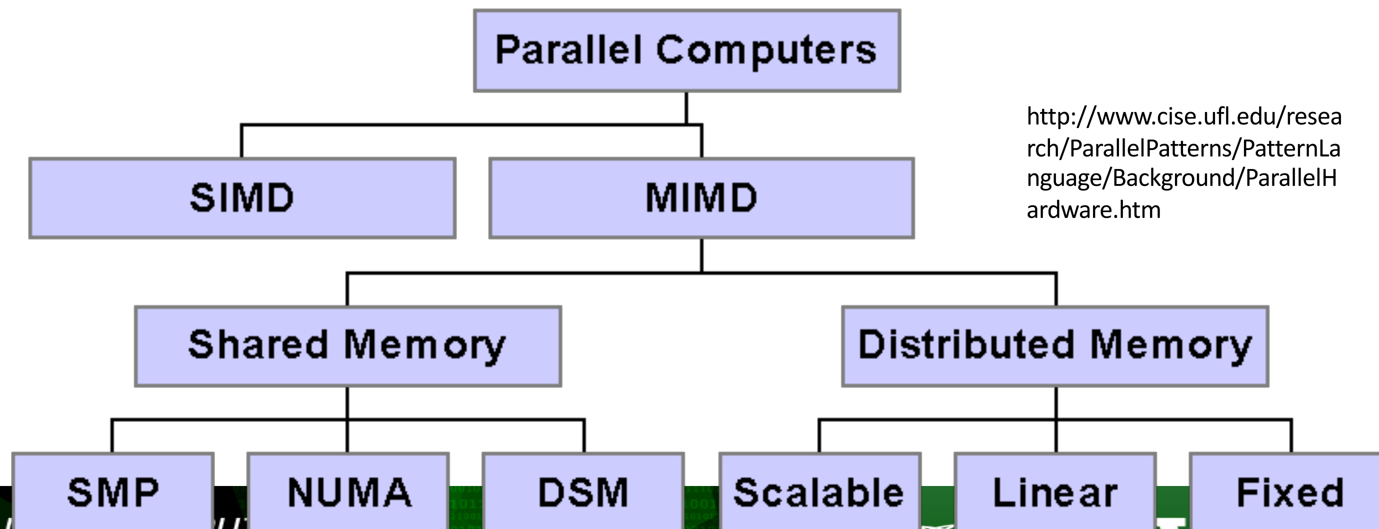


A More Refined (Programmer-Oriented) Taxonomy

- Three major mod uted Memory
- Different program sociated with different modes c MPI for distributed)
- A modern supercomputer will have all three major modes present

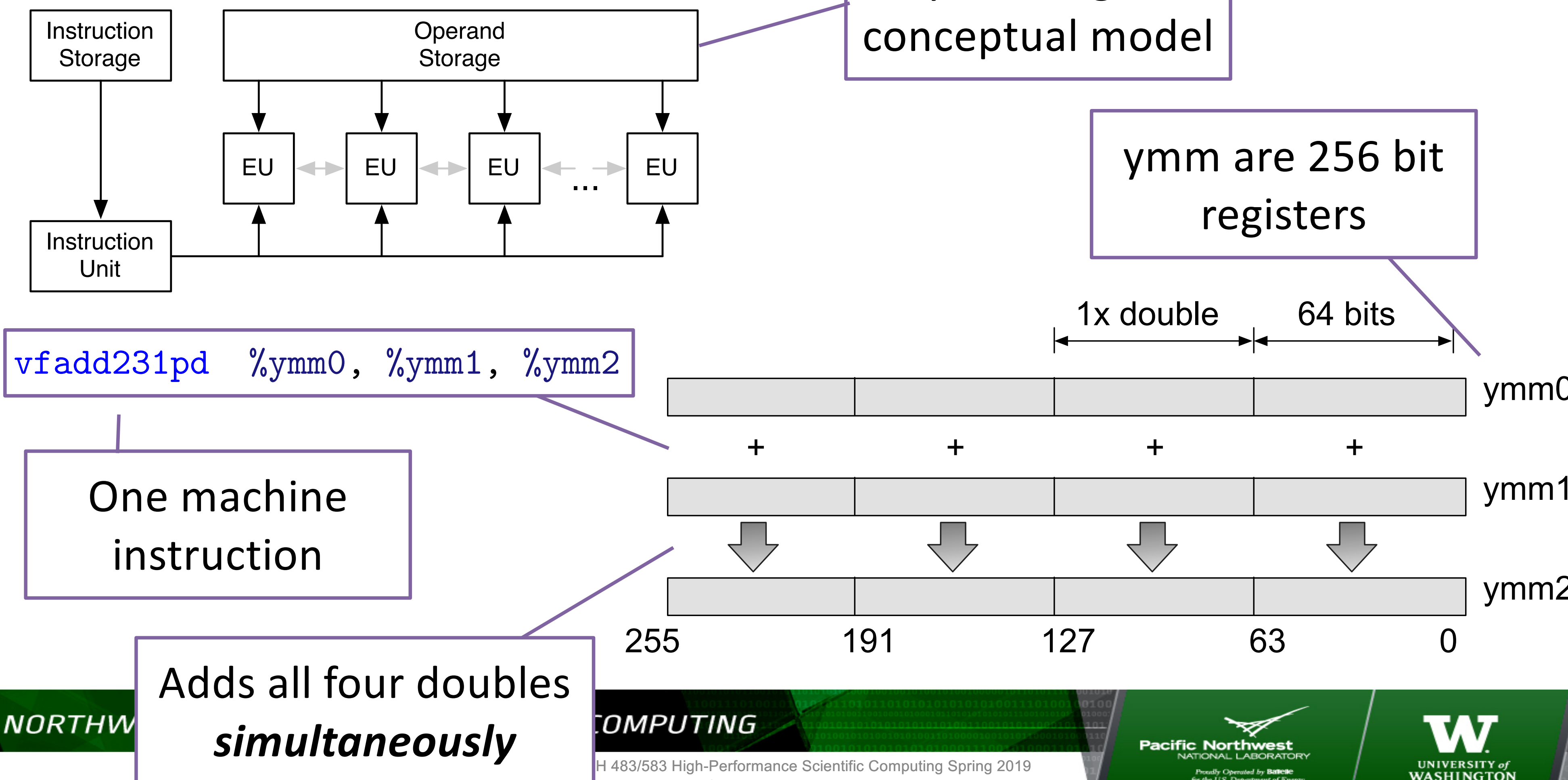
We will come back to this soon



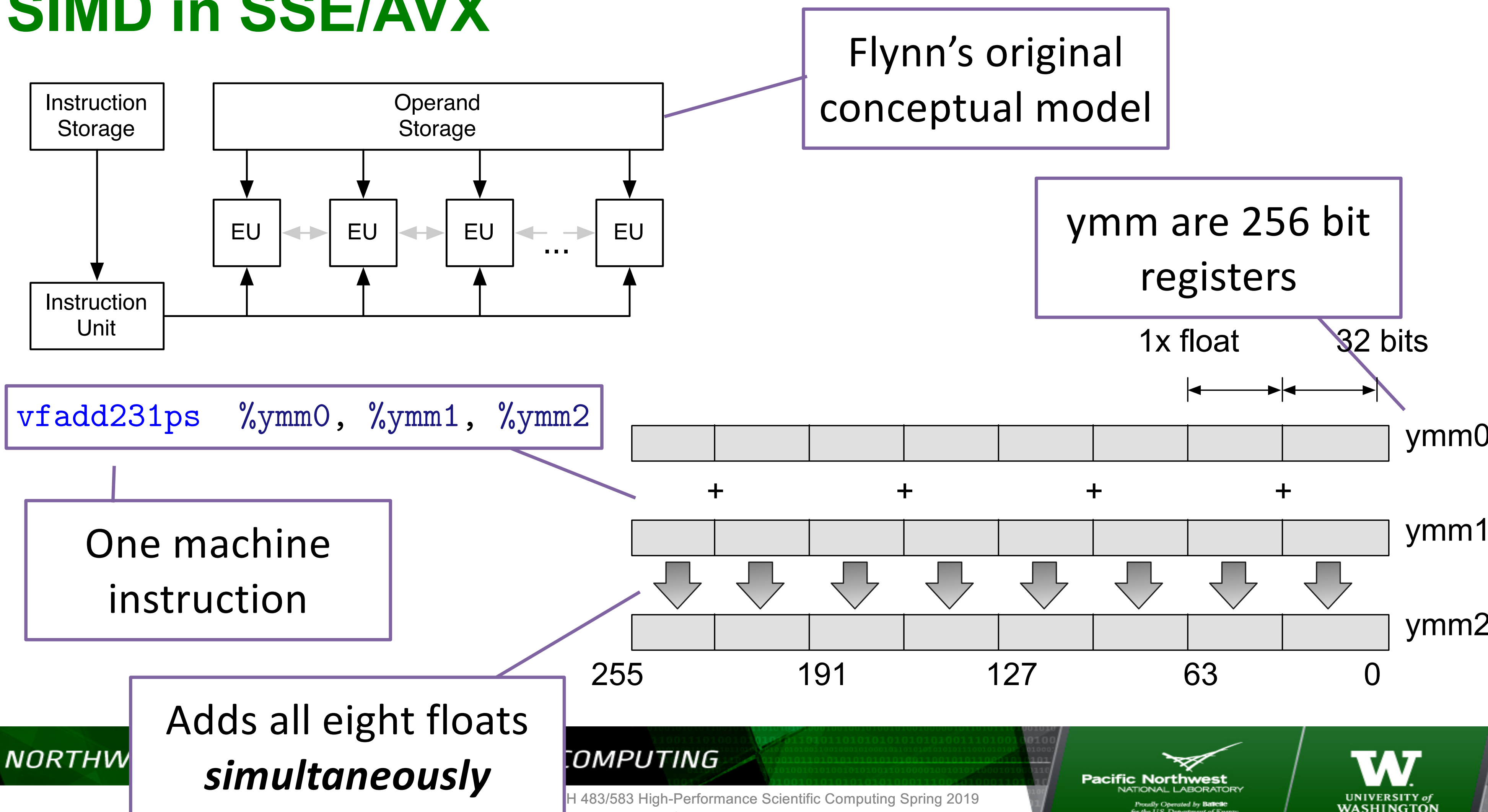


http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ParallelHardware.htm

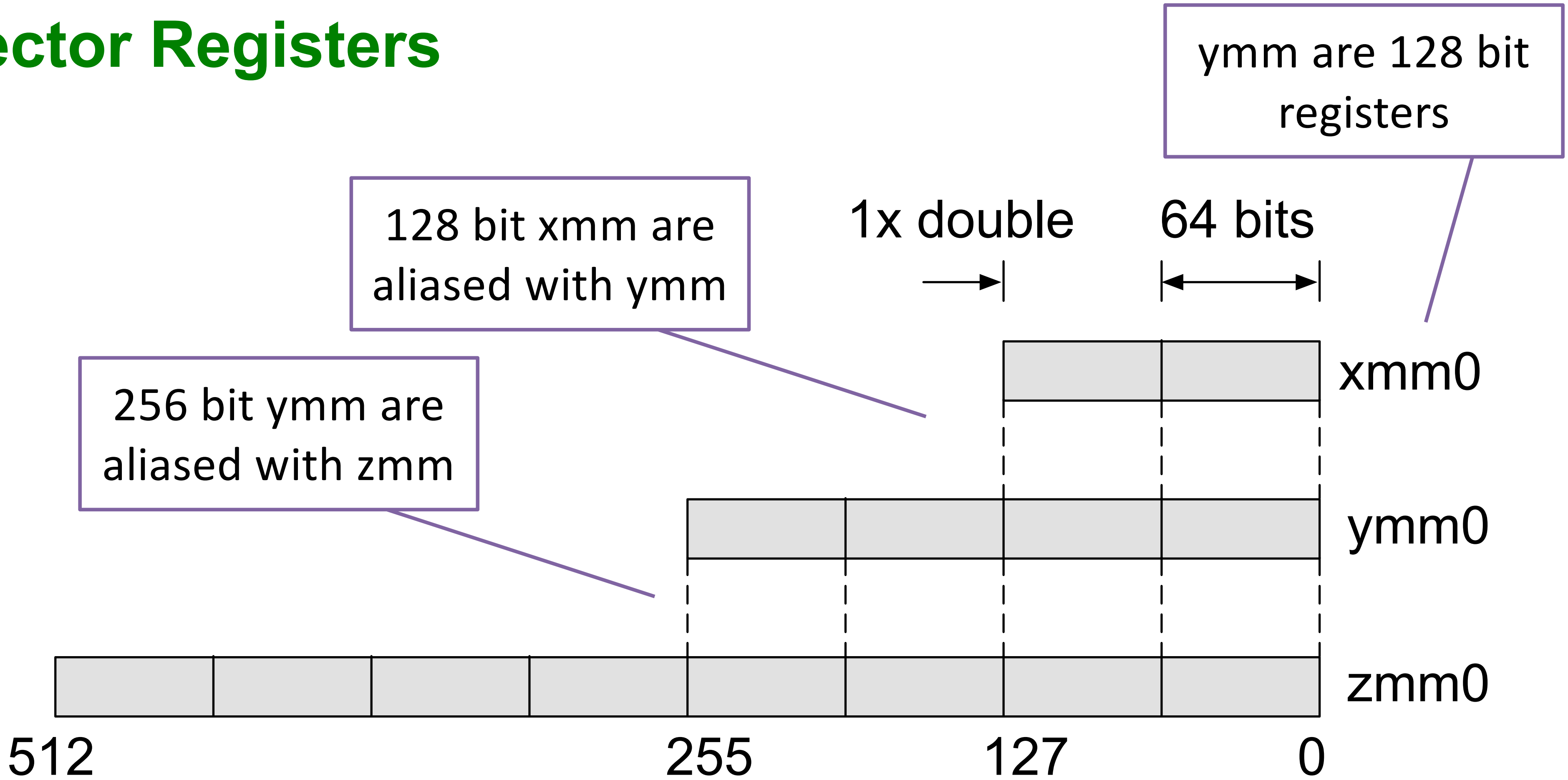
SIMD in SSE/AVX



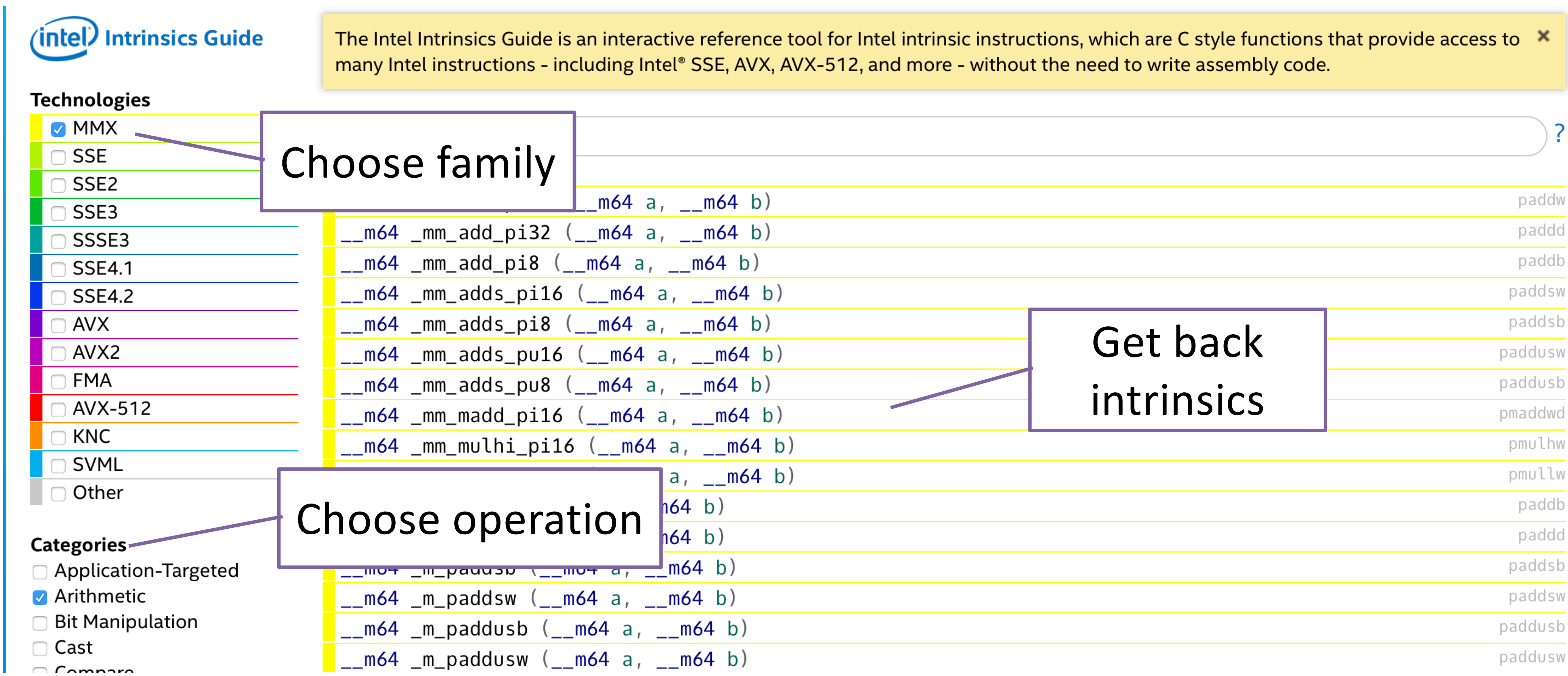
SIMD in SSE/AVX



Vector Registers



Intel Intrinsics Guide



Intrinsics

`__m512d _mm512_fnmadd_pd (__m512d a, __m512d b, __m512d c)` vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Synopsis

```
__m512d _mm512_fnmadd_pd (__m512d a, __m512d b, __m512d c)
#include "immintrin.h"
Instruction: vfnmadd132pd zmm {k}, zmm, zmm
             vfnmadd213pd zmm {k}, zmm, zmm
             vfnmadd231pd zmm {k}, zmm, zmm
CPUID Flags: AVX512F for AVX-512, KNCNI for KNC
```

Description

Multiply packed double-precision (64-bit) floating-point elements in a and … c, and store the results in dst.

Operation

```
FOR j := 0 to 7
        i := j*64
        dst[i+63:i] := -(a[i+63:i] * b[i+63:i]) + c[i+63:i]
ENDFOR
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput
Knights Landing	6	0.5

How to access AVX instructions from C/C++

The machine instruction(s) that is/are generated

Does your CPU support this instruction?

CPU ID

- The cpuid machine instruction can be used to query the CPU about what features it supports

```
$ docker run amath583/cpuinfo
This CPU supports CPUID_EAX_CORE2_DUO_8K
This CPU supports CPUID_EBX_AVX2
This CPU supports CPUID_ECX_SSE3
This CPU supports CPUID_ECX_SSSE3
This CPU supports CPUID_ECX_FMA
This CPU supports CPUID_ECX_SSE41
This CPU supports CPUID_ECX_SSE42
This CPU supports CPUID_ECX_AES
This CPU supports CPUID_ECX_AVX
This CPU supports CPUID_ECX_F16C
This CPU supports CPUID_ECX_HYPERVISOR
```

Processor family

Supported features

Under docker the cpu will be in hypervisor mode

Issuing ASM directly

```
int input = 0, output = 0;

__asm__("cpuid;"
        :"=a"(output)
        :"a"(input)
        :"%ebx","%ecx","%edx"); // clobbered registers
```

C++ variables

cpuid instruction

The register EAX is mapped to variable "output" on completion

The variable "input" is mapped to register EAX at start

Preserve these registers

What Does the Compiler Look for?

```
void basicMultiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (int i = 0; i < A.numRows(); ++i) {
    for (int j = 0; j < B.numCols(); ++j) {
      for (int k = 0; k < A.numCols(); ++k) {
        C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
```

```
Matrix.cpp:31:7: remark: unrolled loop by a factor of 4 \
    with run-time trip count [-Rpass=loop-unroll]

for (int k = 0; k < A.numCols(); ++k) {
```

Unrolling

```
void basicMultiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (int i = 0; i < A.numRows(); ++i) {
    for (int j = 0; j < B.numCols(); ++j) {
      for (int k = 0; k < A.numCols(); k += 4) {
        C(i,j) += A(i, k + 0) * B(k + 0, j);
        C(i,j) += A(i, k + 1) * B(k + 1, j);
        C(i,j) += A(i, k + 2) * B(k + 2, j);
        C(i,j) += A(i, k + 3) * B(k + 3, j);
      }
    }
  }
}
```

Generated Code

```
vmovsd      (%rdi,%r11,8), %xmm1
vmulsd      -8(%r13), %xmm1, %xmm1
vaddsd      %xmm1, %xmm0, %xmm0
vmovsd      %xmm0, (%rdx,%r14,8)
vmovsd      (%r10,%rdi), %xmm1
vmulsd      (%r13), %xmm1, %xmm1
vaddsd      %xmm1, %xmm0, %xmm0
vmovsd      %xmm0, (%rdx,%r14,8)
```

What Does the Compiler Look for?

```
void hoistedMultiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (int i = 0; i < A.numRows(); ++i) {
    for (int j = 0; j < B.numCols(); ++j) {
      double t = C(i,j);
      for (int k = 0; k < A.numCols(); ++k) {
        t += A(i,k) * B(k,j);
      }
      C(i,j) = t;
    }
  }
}
```

```
Matrix.cpp:52:7: remark: vectorized loop \
    (vectorization width: 4, interleaved count: 4) [-Rpass=

      for (int k = 0; k < A.numCols(); ++k) {
      ^
Matrix.cpp:52:7: remark: unrolled loop by a factor of 2 \
    with run-time trip count [-Rpass=loop-unroll]
Matrix.cpp:50:5: remark: unrolled loop by a factor of 8 \
    with run-time trip count [-Rpass=loop-unroll]

    for (int j = 0; j < B.numCols(); ++j) {
    ^
```

What Does the Compiler Look for?

```
./Matrix.hpp:26:69: remark: _ZNKSt3__16vectorIdNS_9allocatorIdEEEixEm inlined into
      _ZNK6MatrixclEmm [-Rpass=inline]
  const double &operator()(size_type i, size_type j) const { return arrayData[i*jCols + j];
                                                                     ^
./Matrix.hpp:25:69: remark: _ZNSt3__16vectorIdNS_9allocatorIdEEEixEm inlined into
      _ZN6MatrixclEmm [-Rpass=inline]
        double &operator()(size_type i, size_type j)       { return arrayData[i*jCols + j];
```

Signatures get mangled

operator()() Function

Function call is replaced with body of code

Easier if body is available to compiler

i.e., if it is ***defined in the header file***

```
vmovsd      (%rdi,%r11,8), %xmm1
vmulsd      -8(%r13), %xmm1, %xmm1
vaddsd      %xmm1, %xmm0, %xmm0
vmovsd      %xmm0, (%rdx,%r14,8)
vmovsd      (%r10,%rdi), %xmm1
vmulsd      (%r13), %xmm1, %xmm1
vaddsd      %xmm1, %xmm0, %xmm0
vmovsd      %xmm0, (%rdx,%r14,8)
```

No function call!!

Without Inlining

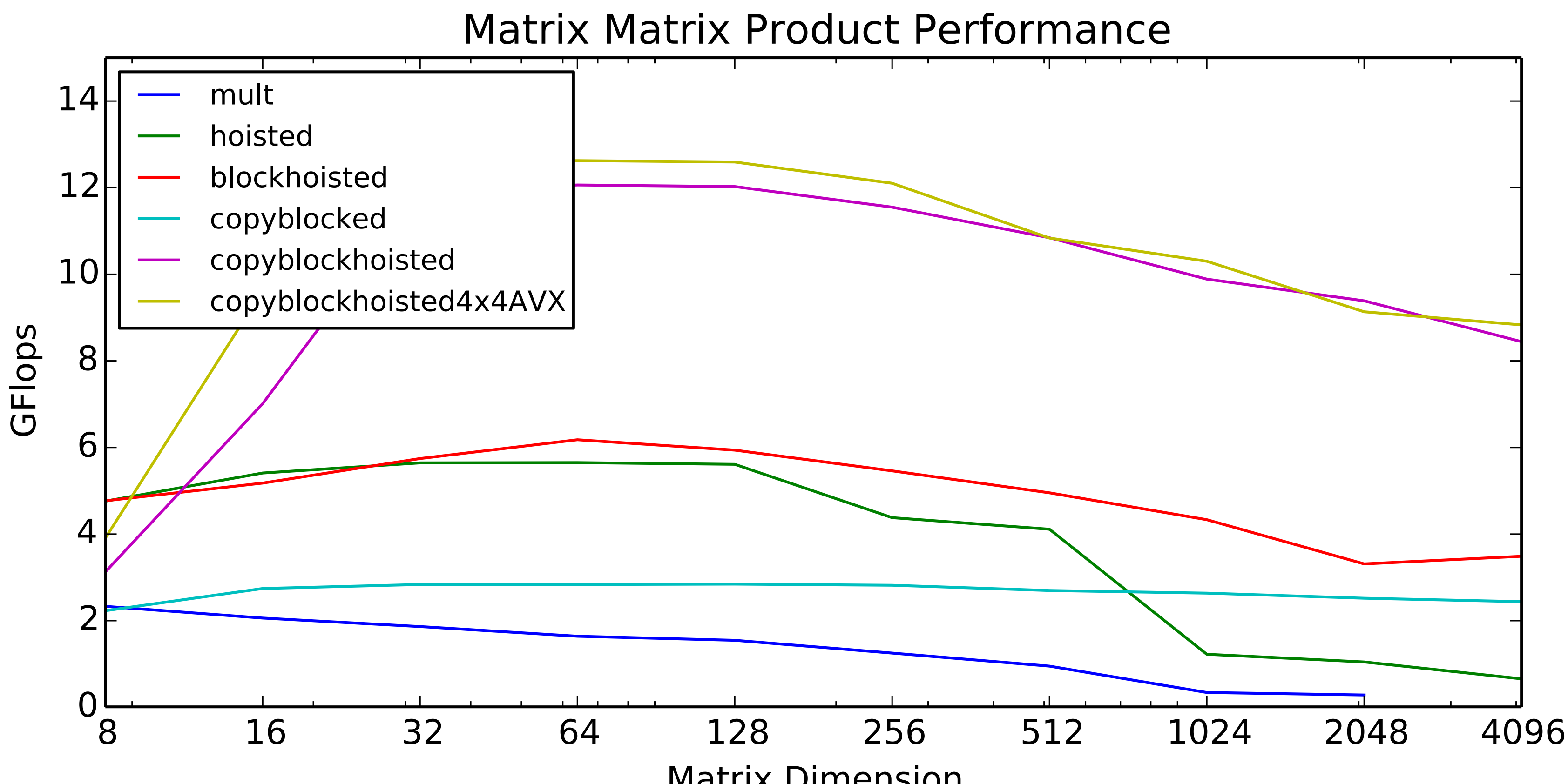
```
movq    -8(%rbp), %rdi
movslq  -28(%rbp), %rsi
movslq  -36(%rbp), %rdx
callq   __ZNK6MatrixclEmm
movsd   (%rax), %xmm0
movq    -16(%rbp), %rdi
movslq  -36(%rbp), %rsi
movslq  -32(%rbp), %rdx
movsd   %xmm0, -72(%rbp)
callq   __ZNK6MatrixclEmm
movsd   -72(%rbp), %xmm0
mulsd   (%rax), %xmm0
movq    -24(%rbp), %rdi
movslq  -28(%rbp), %rsi
movslq  -32(%rbp), %rdx
movsd   %xmm0, -80(%rbp)
callq   __ZN6MatrixclEmm
movsd   -80(%rbp), %xmm0
addsd   (%rax), %xmm0
movsd   %xmm0, (%rax)
```

operator()() function call

operator()() function call

operator()() function call

Summary


Matrix Matrix Product Performance
mult
hoisted
blockhoisted
copyblocked
copyblockhoisted
copyblockhoisted4x4AVX
GFlops
0
2
4
6
8
10
12
14
8
16
32
64
128
256
512
1024
2048
4096
Matrix Dimension

Recommendations

Inlining, unrolling, vectorization

- Avoid programming in assembler
- If you can't avoid that, use intrinsics – but you will need to match the instructions to the hardware (which is not portable)
- In general, let compiler determine hardware, pick instructions, and optimize
- Check your performance against performance models
- Monitor what your compiler is doing
  - Optimization report
  - Full set of flags
  - Last resort – read the assembler

Most important is to have a mental model for the vector registers and to be aware of what is possible and how to write code to be optimizable

Review

- High Performance = Writing software to use hardware effectively
- Hardware
  - Fast clock
  - Branch prediction, other magic on chip
  - Hierarchical memory
  - Pipelining instructions
  - Vector registers and vector instructions ("SIMD")
- Software techniques to use all of these
- Compilers!
- Our first parallel computations

Tuning

- Starting with base code
- Various compiler optimizations help
- Tiling (which size)
- Blocking (what size)
- What size works best for Tiling and Blocking ***together?***

- What loop ordering? Matrix matrix product has six different orderings? What block ordering?

- What about when we add AVX, and threads, etc?

How do we find the optimal combination?

Magic: the power of apparently influencing the course of events by using mysterious or supernatural forces

The answer will be different for different CPUs

Finding the Sweet Spot

- Exhaustive parameter space search
  - Tiling, Blocking, Compiler flags, AVX inst, loop ordering
- Original project at UC Berkeley phiPAC (Bilmes et al)
- Further developed by Whaley and Dongarra → Automatically Tuned Linear Algebra Subprograms (ATLAS)
  - Recently honored with "test of time" award

And wrote a program to generate different multiply functions

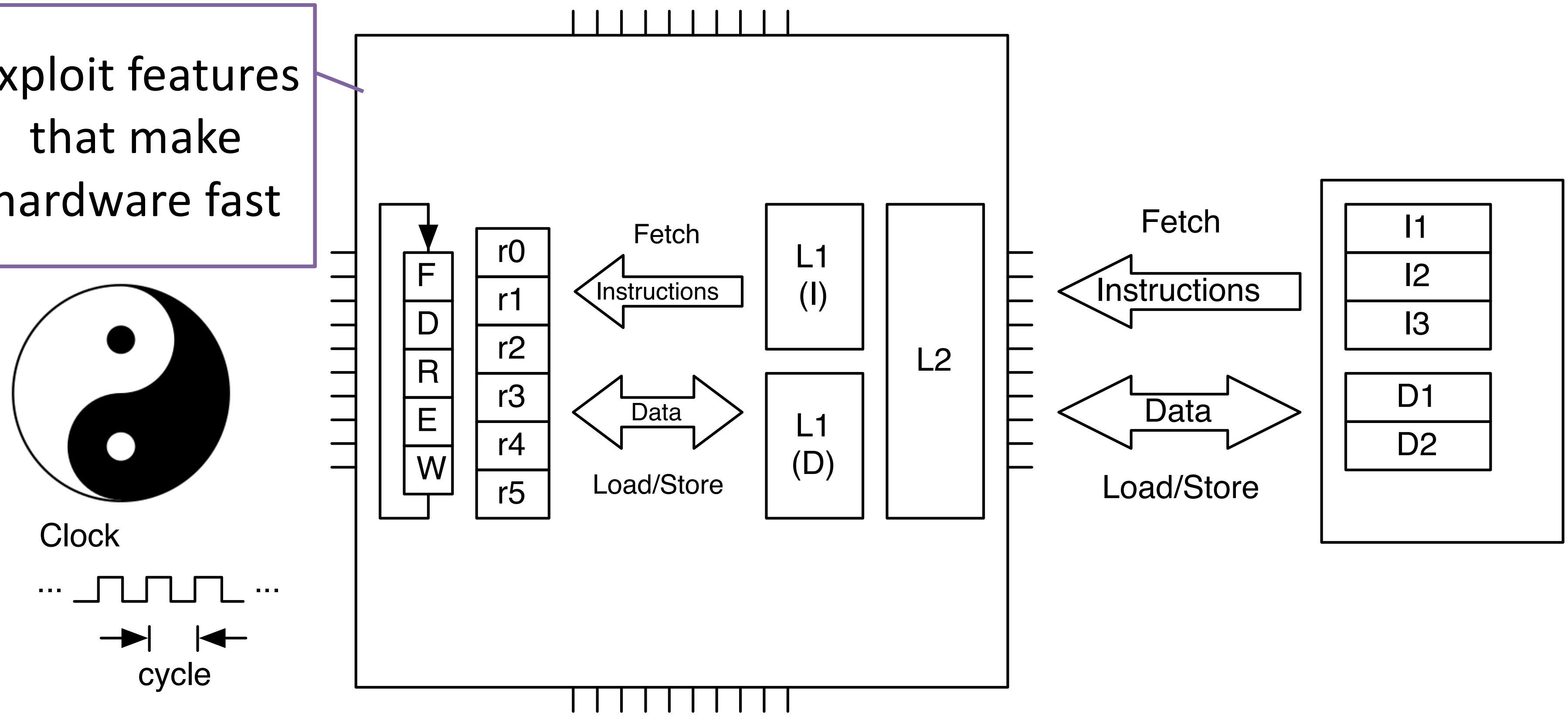
This started as a final course project

The competition was to write fastest matrix-matrix product

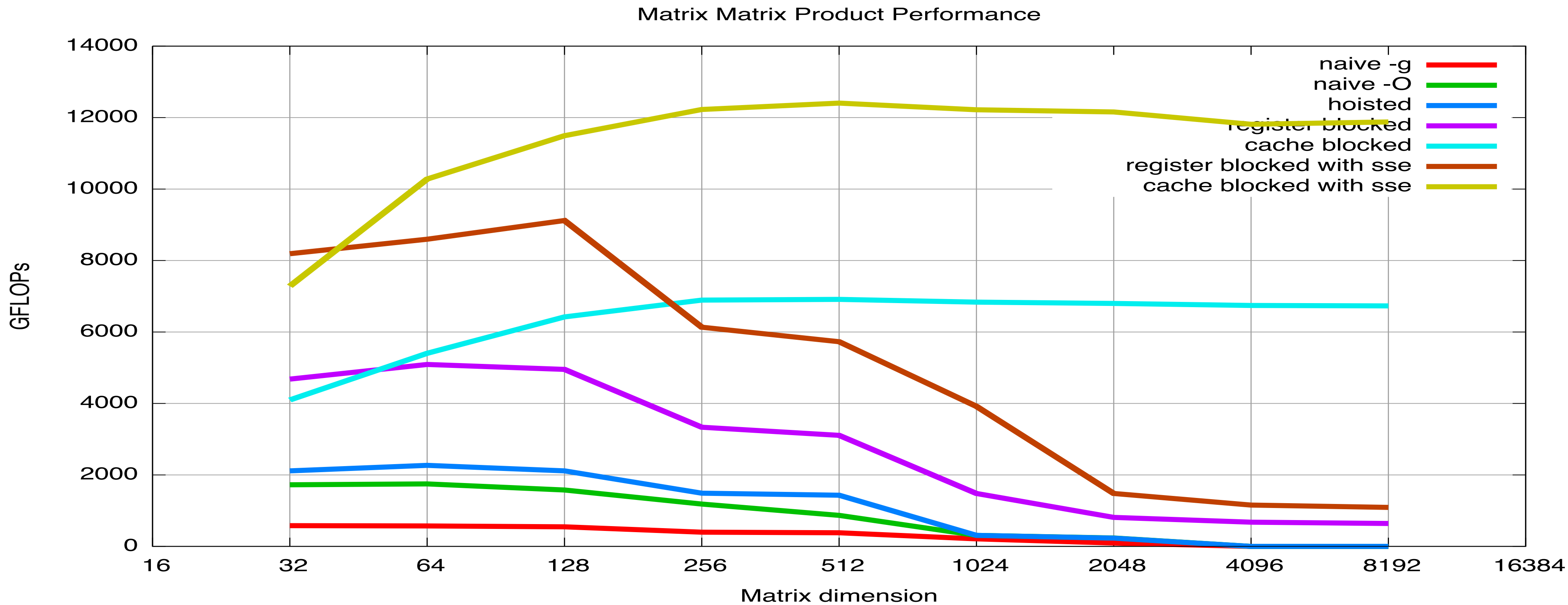
Students were the good kind of lazy

What Else Can We Do for Performance

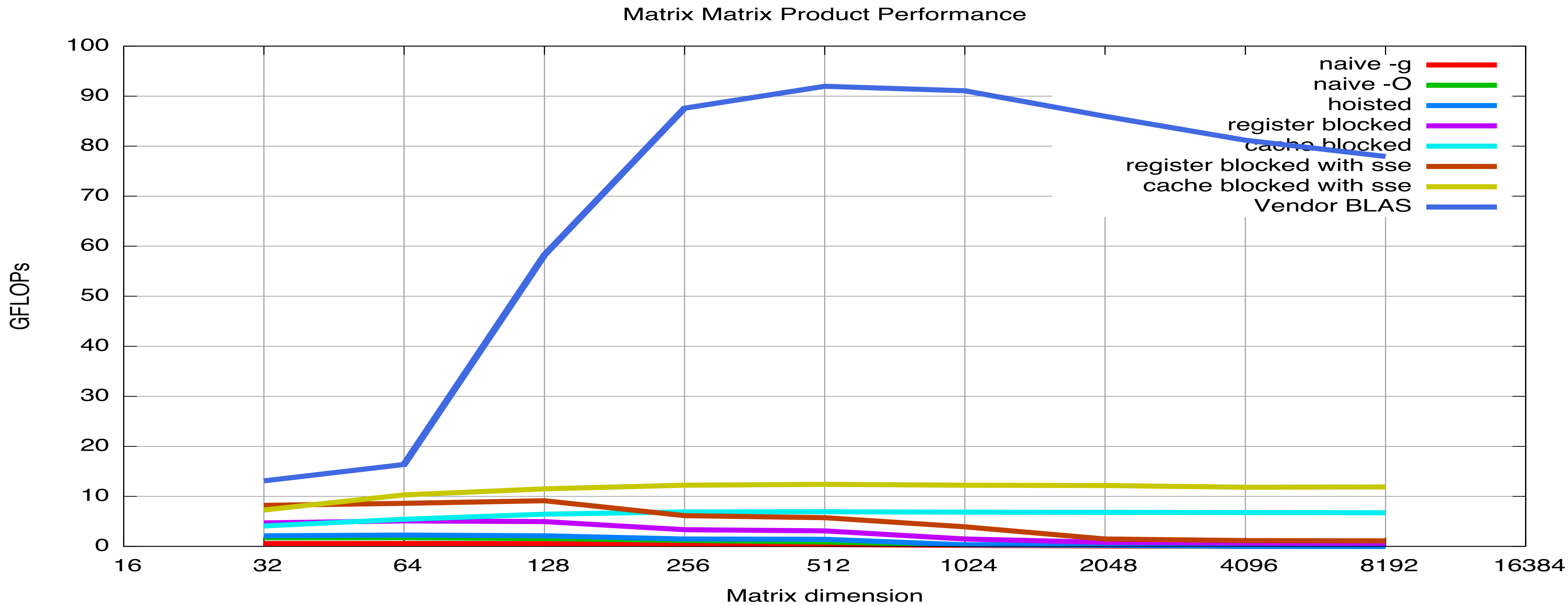
Exploit features that make hardware fast



Example: SIMD Matrix-Matrix Product



Example: Vendor Supplied Matrix-Matrix Product



Basic Linear Algebra Subprograms (BLAS)

- Standardized set of core / kernel algorithms for numerical linear algebra
- Fortran – but various extensions to C have been created
  - Matrix ordering and function calling disciplines are main Fortran/C issues
- Originally derived from needs of LINPACK / EISPACK then LAPACK
- Four precisions: single, double, single complex, double complex
  - "s", "d", "c", "z" prefixes
- Level-1: Vector-vector operations
  - Double precision vector addition = "daxpy"
- Level-2: Matrix-vector operations
  - Double precision matrix-vector product = "dgemv"
- Level-3: Matrix-matrix operations
  - Double precision matrix-matrix product = "dgemm"
- There are also sparse BLAS and a next-generation BLAS, but neither are well-supported by vendors

BLAS

- (Updated set of) Basic Linear Algebra Subprograms

- The BLAS functionality is divided into three levels:
  - **Level 1:** contains vector operations of the form:

    $$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$$

    as well as scalar dot products and vector norms

  - **Level 2:** contains matrix-vector operations of the form

    $$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

    as well as Tx = y solving for x with T being triangular

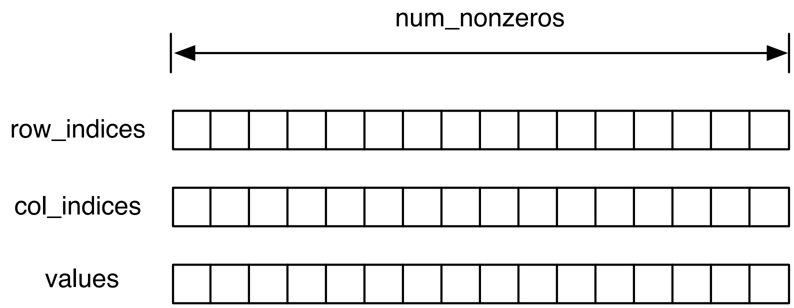
  - **Level 3:** contains matrix-matrix operations of the form

    $$C \leftarrow \alpha AB + \beta C$$

    as well as solving $B \leftarrow \alpha T^{-1} B$ for triangular matrices T. This level contains the widely used General Matrix Multiply operation.

Coordinate Storage

- Store each element row number, column number, and value




```
size_t num_nonzeros;
index_t *row_indices;
index_t *column_indices;
double  *values;

row_indices = malloc(num_nonzeros * sizeof(index_t));
column_indices = malloc(num_nonzeros * sizeof(index_t));
values = malloc(num_nonzeros * sizeof(double));

for (int k = 0; k < num_nonzeros; ++k)
  y[column_indices[k]] += values[k] * y[row_indices[k]];
```

BLAS

- Several implementations for different languages exist
  - Reference implementation (F77 and C-wrapper)
    http://www.netlib.org/blas/
  - ATLAS, highly optimized for particular processor architectures
  - A generic C++ template class library providing BLAS functionality: uBLAS http://www.boost.org
  - Several vendors provide libraries optimized for their architecture (AMD, HP, IBM, Intel, NEC, NViDIA, Sun)

BLAS: F77 naming conventions

- Each routine has a name which specifies the operation, the type of matrices involved and their precisions.
  Names are in the form: PMMOO

  – Some of the most common operations (OO):
    - **DOT** scalar product, x^T y
      AXPY vector sum, α x + y
      MV matrix-vector product, A x
      SV matrix-vector solve, inv(A) x
      MM matrix-matrix product, A B
      SM matrix-matrix solve, inv(A) B

  – The types of matrices are (MM)
    - **GE** general
      GB general band
      SY symmetric
      SB symmetric band
      SP symmetric packed
      HE hermitian
      HB hermitian band
      HP hermitian packed
      TR triangular
      TB triangular band
      TP triangular packed

  – Each operation is defined for four precisions (P)
    - **S** single real
      D double real
      C single complex
      Z double complex

- Examples

  SGEMM stands for "single-precision general matrix-matrix multiply"

  DGEMM stands for "double-precision matrix-matrix multiply".

BLAS: C naming conventions

- F77 routine name is changed to lowercase and prefixed with *cblas_*
- All routines accepting two dimensional arrays have a new additional first parameter specifying the matrix memory layout (row major or column major)
- Character parameters are replaced by corresponding enum values
- Input arguments are declared **`const`**
- Non-complex scalar input parameters are passed by value
- Complex scalar input arguments are passed using a **`void*`**
- Arrays are passed by address
- Output scalar arguments are passed by address
- Complex functions become subroutines which return the result via an additional last parameter (**`void*`**), appending *_sub* to the name

_axpy

cblas_daxpy

Computes a constant times a vector plus a vector (double-precision).

```
void cblas_daxpy (
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   double *Y,
   const int incY
);
```

Parameters

N
Number of elements in the vectors.

alpha
Scaling factor for the values in `X`.

X
Input vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
Input vector `Y`.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

On return, the contents of vector Y are replaced with the result. The value computed is `(alpha * X[i]) + Y[i]`.

Availability

Available in OS X v10.2 and later.

Declared In

`cblas.h`

cblas_dgemm

cblas_dgemm

Multiplies two matrices (double-precision).

```
void cblas_dgemm (
    const enum CBLAS_ORDER Order,
    const enum CBLAS_TRANSPOSE TransA,
    const enum CBLAS_TRANSPOSE TransB,
    const int M,
    const int N,
    const int K,
    const double alpha,
    const double *A,
    const int lda,
    const double *B,
    const int ldb,
    const double beta,
    double *C,
    const int ldc
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
Specifies whether to transpose matrix `A`.

TransB
Specifies whether to transpose matrix `B`.

M
Number of rows in matrices `A` and `C`.

N
Number of columns in matrices `B` and `C`.

K
Number of columns in matrix `A`; number of rows in matrix `B`.

alpha
Scaling factor for the product of matrices A and B.

A
Matrix A.

lda
The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
Matrix B.

ldb
The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
Scaling factor for matrix C.

C
Matrix C.

ldc
The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Basic Linear Algebra Subprograms (BLAS)

- Level 1: Vector-Vector operations

name	description	equation	prefixes
_rotg	generate plane rotation		s, d
_rotmg	generate modified plane rotation		s, d
_rot	apply plane rotation		s, d
_rotm	apply modified plane rotation		s, d
_swap	swap vectors	$x \leftrightarrow y$	s, d, c, z
_scal	scale vector	$y = \alpha y$	s, d, c, z, cs, zd
_copy	copy vector	$y = x$	s, d, c, z
_axpy	update vector	$y = y + \alpha x$	s, d, c, z
_dot	dot product	$= x^t y$	s, d, ds
_dotc	complex conj dot	$= x^h y$	c, z
_dotu	complex dot	$= x^t y$	c, z
__dot		$= \alpha + x^t y$	sds
_nrm2	2-norm	$= \|x\|_2$	s, d, sc, dz
_asum	1-norm	$= \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$	s, d, sc, dz
i_amax	∞-norm	$= i$ such that $\lvert\mathrm{Re}(x_i)\rvert + \lvert\mathrm{Im}(x_i)\rvert$ is max	s, d, c, z

name	description	equation	prefixes
_gemv	general matrix-vector multiply	$y = \alpha A^* x + \beta y$	s, d, c, z
_gbmv	(banded)	$y = \alpha A^* x + \beta y$	s, d, c, z
_hemv	hermetian mat-vec	$y = \alpha A x + \beta y$	c, z
_hbmv	(banded)	$y = \alpha A x + \beta y$	c, z
_hpmv	(packed)	$y = \alpha A x + \beta y$	c, z
_symv	symmetric mat-vec	$y = \alpha A x + \beta y$	s, d, (c, z)†
_sbmv	(banded)	$y = \alpha A x + \beta y$	s, d
_spmv	(packed)	$y = \alpha A x + \beta y$	s, d, (c, z)†
_trmv	triangular mat-vec	$x = A^* x$	s, d, c, z
_tbmv	(banded)	$x = A^* x$	s, d, c, z
_tpmv	(packed)	$x = A^* x$	s, d, c, z
_trsv	triangular solve	$x = A^{-*} x$	s, d, c, z
_tbsv	(banded)	$x = A^{-*} x$	s, d, c, z
_tpsv	(packed)	$x = A^{-*} x$	s, d, c, z

A^* denotes A, A^T, or A^H;
A^{-*} denotes A^{-1}, A^{-T}, or A^{-H}, depending on options and data type.
A is $m \times n$ or $n \times n$

name	description	equation	prefixes
_ger	general rank-1 update	$A = A + \alpha x y^T$	s, d
_geru	(complex)	$A = A + \alpha x y^T$	c, z
_gerc	(complex conj)	$A = A + \alpha x y^H$	c, z
_her	hermetian rank-1 update	$A = A + \alpha x x^H$	c, z
_hpr	(packed)	$A = A + \alpha x x^H$	c, z
_her2	hermetian rank-2 update	$A = A + \alpha x y^H + y(\alpha x)^H$	c, z
_hpr2	(packed)	$A = A + \alpha x y^H + y(\alpha x)^H$	c, z
_syr	symmetric rank-1 update	$A = A + \alpha x x^T$	s, d, (c, z)†
_spr	(packed)	$A = A + \alpha x x^T$	s, d, (c, z)†
_syr2	symmetric rank-2 update	$A = A + \alpha x y^T + \alpha y x^T$	s, d
_spr2	(packed)	$A = A + \alpha x y^T + \alpha y x^T$	s, d

name	description	equation	prefixes
_gemm	general matrix-matrix multiply	$C = \alpha A^* B^* + \beta C$	s, d, c, z
_symm	symmetric mat-mat	$C = \alpha AB + \beta C$	s, d, c, z
_hemm	hermetian mat-mat	$C = \alpha AB + \beta C$	c, z
_syrk	symmetric rank-k update	$C = \alpha AA^T + \beta C$	s, d, c, z
_herk	hermetian rank-k update	$C = \alpha AA^H + \beta C$	c, z
_syr2k	symmetric rank-$2k$ update	$C = \alpha AB^T + \bar{\alpha} BA^T + \beta C$	s, d, c, z
_her2k	hermetian rank-$2k$ update	$C = \alpha AB^H + \bar{\alpha} BA^H + \beta C$	c, z
_trmm	triangular mat-mat	$B = \alpha A^* B$ or $B = \alpha BA^*$	s, d, c, z
_trsm	triangular solve mat	$B = \alpha A^{-*} B$ or $B = \alpha BA^{-*}$	s, d, c, z

A^* denotes A, A^T, or A^H;
A^{-*} denotes A^{-1}, A^{-T}, or A^{-H}, depending on options and data type.
The destination matrix is $m \times n$ or $n \times n$. For mat-mat, the common dimension of A and B is k.

Calling Fortran from C

- DGEMM in Fortran

```
      SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
     $                   BETA, C, LDC )
#     Scalar Arguments
      CHARACTER*1        TRANSA, TRANSB
      INTEGER            M, N, K, LDA, LDB, LDC
      DOUBLE PRECISION   ALPHA, BETA
#     Array Arguments
      DOUBLE PRECISION   A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- Corresponding C prototype

```
void dgemm_(const char* transa, const char* transb,
      const int* m, const int* n, const int* k,
      const double* alpha,  const double *da,  const int*
         lda,
      const double *db, const int* ldb, const double*
         dbeta,
      double *dc, const int* ldc);
```

Calling Fortran from C

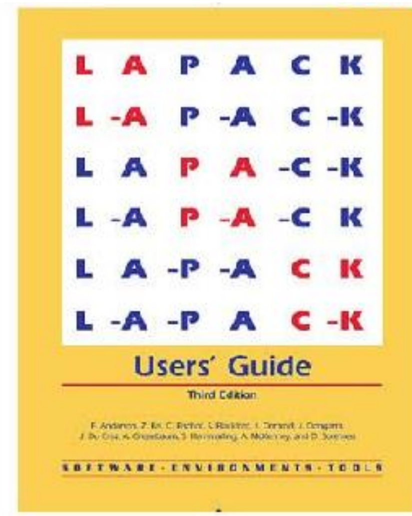
- Corresponding C prototype

```
void dgemm_(const char* transa, const char* transb,
      const int* m, const int* n, const int* k,
      const double* alpha,  const double *da,  const int*
         lda,
      const double *db, const int* ldb, const double*
         dbeta,
      double *dc, const int* ldc);
```

- What other issues do you need to be aware of when calling Fortran dgemm?

LAPACK

- F77, based on blocked algorithms (BLAS 3)
- Driver routines (simple and expert) used to solve a complete problem
  - Solve a linear system of equations
  - Least squares solutions
  - Eigenvalue problems
  - Singular value problems
- Routines for distinct computational tasks
  - LU / Cholesky / QR / SVD factorization
  - Schur / generalized Schur decomposition
- Auxiliary
  - Estimate condition numbers
  - Unblocked algorithms
  - Future extensions to BLAS



http://www.netlib.org/lapack

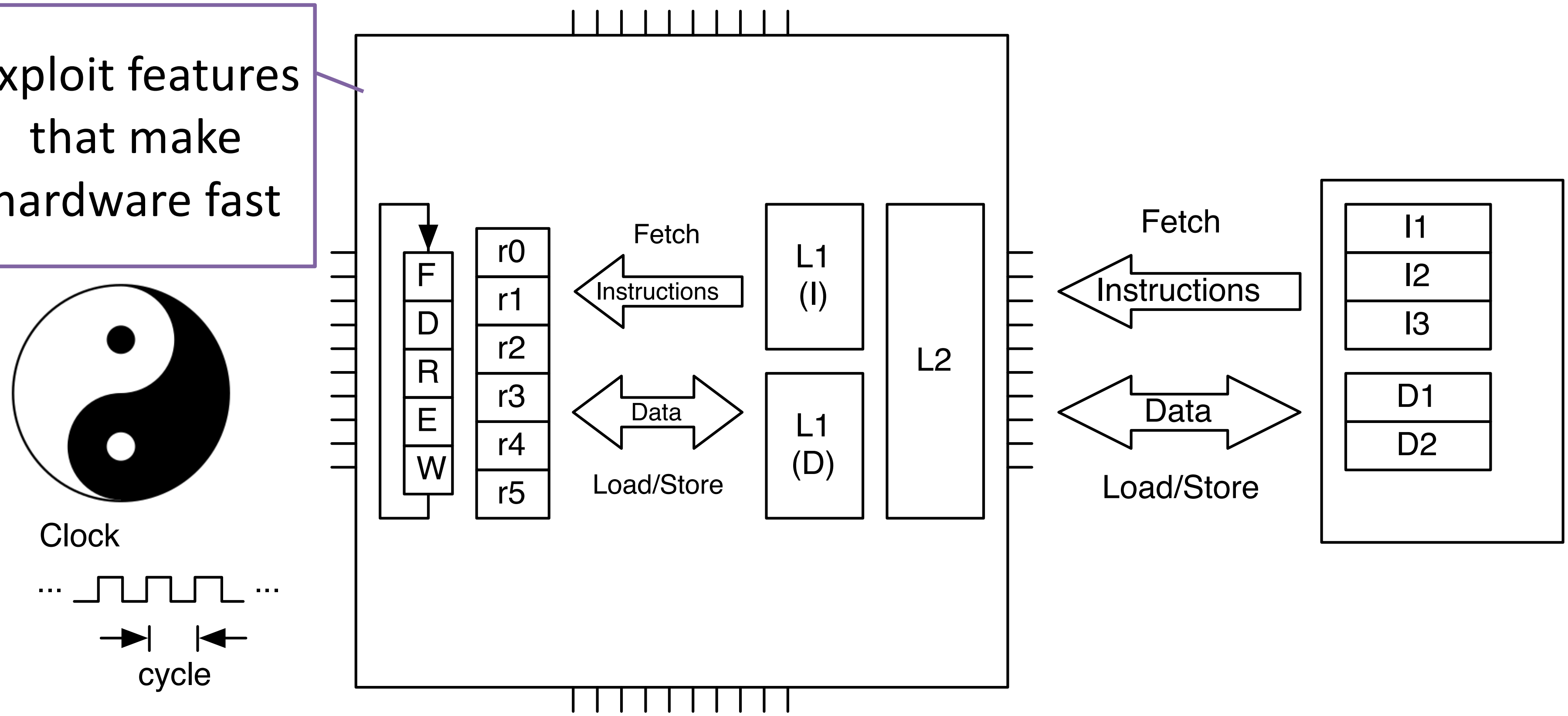
LAPACK naming conventions

- Similar to BLAS
  - **XYYZZZ**
    - **X**: data type
      - **S**: REAL
      - **D**: DOUBLE PRECISION
      - **C**: COMPLEX
      - **Z**: COMPLEX*16 or DOUBLE COMPLEX
    - **YY**: matrix type
      - **BD**: bidiagonal
      - **DI**: diagonal
      - **GB**: general band
      - **GE**: general (i.e., unsymmetric, in some cases rectangular)
      - **GG**: general matrices, generalized problem (i.e., a pair of general matrices)
      - **GT**: general tridiagonal
      - **HB**: (complex) Hermitian band
      - **HE**: (complex) Hermitian
      - **HG**: upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a triangular matrix)
      - **HP**: (complex) Hermitian, packed storage
      - **HS**: upper Hessenberg
      - **OP**: (real) orthogonal, packed storage
      - **OR**: (real) orthogonal
      - **PB**: symmetric or Hermitian positive definite band

- **YY**: more matrix types
  - **PO**: symmetric or Hermitian positive definite
  - **PP**: symmetric or Hermitian positive definite, packed storage
  - **PT**: symmetric or Hermitian positive definite tridiagonal
  - **SB**: (real) symmetric band
  - **SP**: symmetric, packed storage
  - **ST**: (real) symmetric tridiagonal
  - **SY**: symmetric
  - **TB**: triangular band
  - **TG**: triangular matrices, generalized problem (i.e., a pair of triangular matrices)
  - **TP**: triangular, packed storage
  - **TR**: triangular (or in some cases quasi-triangular)
  - **TZ**: trapezoidal
  - **UN**: (complex) unitary
  - **UP**: (complex) unitary, packed storage
- **ZZZ**: performed computation
  - Linear systems
  - Factorizations
  - Eigenvalue problems
  - Singular value decomposition
  - Etc.

What Else Can We Do for Performance

Exploit features that make hardware fast



General Performance Principles

- Work harder
  - Faster core
- Work smarter
  - Branch predictions, etc
  - Better compilation
  - Better algorithm
  - Better implementation
- Get help

Dennard scaling (ended 2005)

What about this?

We did this

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$\begin{aligned} C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\ C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\ C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\ C_{11} &= A_{10}B_{01} + A_{11}B_{11} \end{aligned}$$

Eight multiplies

If these are matrix blocks: Eight matrix multiplies

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B & \end{bmatrix}$$

Seven matrix multiplies

Seven multiplies

Recurse

$$\begin{aligned}
T_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
T_1 &= (A_{10} + A_{11})(B_{00}) \\
T_2 &= (A_{00})(B_{01} - B_{11}) \\
T_3 &= (A_{11})(B_{10} - B_{00}) \\
T_4 &= (A_{00} + A_{01})(B_{11}) \\
T_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\
T_6 &= (A_{01} - A_{11})(B_{10} + B_{11})
\end{aligned}$$

$$\begin{aligned}
C_{00} &= T_0 + T_3 - T_4 + T_6 \\
C_{01} &= T_2 + T_4 \\
C_{10} &= T_1 + T_4 \\
C_{11} &= T_0 - T_1 + T_2 + T_5
\end{aligned}$$

Many adds and subtracts

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Seven matrix multiplies

Recurse

$$\begin{aligned}
T_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
T_1 &= (A_{10} + A_{11})(B_{00}) \\
T_2 &= (A_{00})(B_{01} - B_{11}) \\
T_3 &= (A_{11})(B_{10} - B_{00}) \\
T_4 &= (A_{00} + A_{01})(B_{11}) \\
T_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\
T_6 &= (A_{01} - A_{11})(B_{10} + B_{11})
\end{aligned}$$

$$\begin{aligned}
C_{00} &= T_0 + T_3 - T_4 + T_6 \\
C_{01} &= T_2 + T_4 \\
C_{10} &= T_1 + T_4 \\
C_{11} &= T_0 - T_1 + T_2 + T_5
\end{aligned}$$

$$O(N^3) \text{ work vs } O(N^2) \text{ data}$$

Multiply

Add

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Divide and Conquer

$$\begin{aligned} T_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ T_1 &= (A_{10} + A_{11})(B_{00}) \\ T_2 &= (A_{00})(B_{01} - B_{11}) \\ T_3 &= (A_{11})(B_{10} - B_{00}) \\ T_4 &= (A_{00} + A_{01})(B_{11}) \\ T_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ T_6 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned}$$

→

$$\begin{aligned} C_{00} &= T_0 + T_3 - T_4 + T_6 \\ C_{01} &= T_2 + T_4 \\ C_{10} &= T_1 + T_4 \\ C_{11} &= T_0 - T_1 + T_2 + T_5 \end{aligned}$$

Recurse

Seven matrix multiplies

$O(N^3)$ work vs $O(N^2)$ data

Each block is size $\frac{N}{2}$ → $\left(\frac{N}{2}\right)^3 = \frac{N^3}{8}$ → $\frac{7}{8}N^3$

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$\frac{7}{8}\frac{7}{8}\cdots\frac{7}{8}$$

How many of these

Divide and conquer

$$\begin{aligned}
T_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
T_1 &= (A_{10} + A_{11})(B_{00}) \\
T_2 &= (A_{00})(B_{01} - B_{11}) \\
T_3 &= (A_{11})(B_{10} - B_{00}) \\
T_4 &= (A_{00} + A_{01})(B_{11}) \\
T_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\
T_6 &= (A_{01} - A_{11})(B_{10} + B_{11})
\end{aligned}$$

$$\begin{aligned}
C_{00} &= T_0 + T_3 - T_4 + T_6 \\
C_{01} &= T_2 + T_4 \\
C_{10} &= T_1 + T_4 \\
C_{11} &= T_0 - T_1 + T_2 + T_5
\end{aligned}$$

$$\log_2(N)$$

$$O(N^{\log_2 7})$$

$$O(N^{\log_2 7}) \ll O(N^{\log_2 8}) = O(N^3)$$

Strassen's Algorithm

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$\begin{aligned} T_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ T_1 &= (A_{10} + A_{11})(B_{00}) \\ T_2 &= (A_{00})(B_{01} - B_{11}) \\ T_3 &= (A_{11})(B_{10} - B_{00}) \\ T_4 &= (A_{00} + A_{01})(B_{11}) \\ T_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ T_6 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned}$$

$$\begin{aligned} C_{00} &= T_0 + T_3 - T_4 + T_6 \\ C_{01} &= T_2 + T_4 \\ C_{10} &= T_1 + T_4 \\ C_{11} &= T_0 - T_1 + T_2 + T_5 \end{aligned}$$

$O(N^{2.38})$

Better algorithms

Limit?

Require large N

Limit Unknown, Biggest open question in numerical linear algebra

Thank You!