AMATH 483/583
High Performance Scientific Computing

Lecture 7: Compilation, optimization, SIMD/Vector

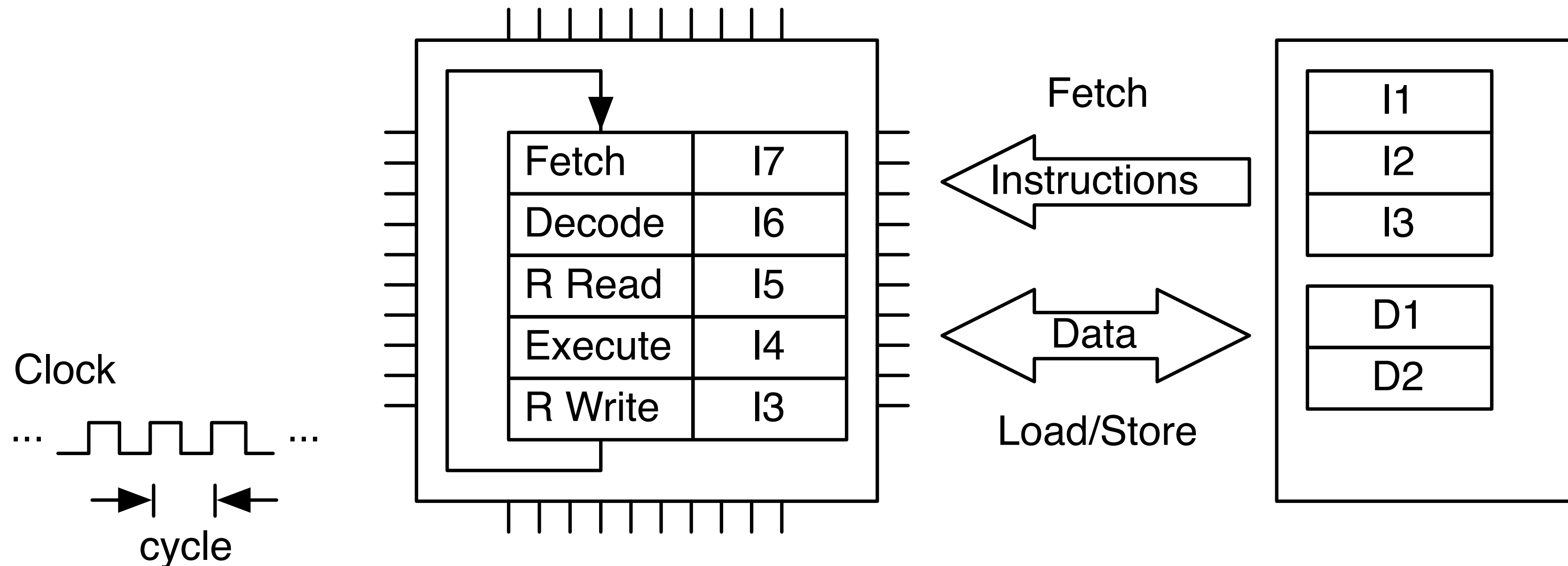
Xu Tony Liu, PhD
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA

Overview

- Brief review of optimization techniques
- Improve locality
  - Hoisting
  - Unrolling
  - Blocking (also known as tiling)
  - Copying
- 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

Separate L1 and L2 for each core

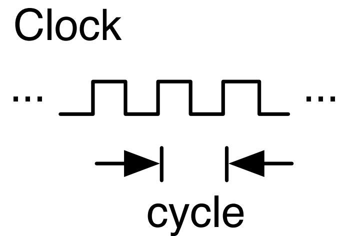
Cores work on separate register sets and instrs

Shared L3

Cores work on separate register sets and instrs

F
D
R
E
W
r0
r1
r2
r3
r4
r5
Fetch
Instructions
Data
Load/Store
L1 (I)
L1 (D)
L2
L3

Fetch
Instructions
Data
Load/Store
I1
I2
I3
D1
D2



Main memory is shared

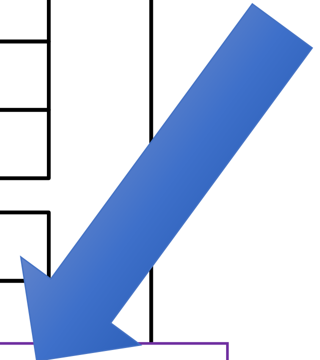
Locality → Strategy

The next operand may be "near" the last

It could be "near" in time or space

Near in time (***temporal locality***): the next operand is a previous operand

Near in space (***spatial locality***): the next operand is in a nearby memory location to a previous operand



F
D
R
E
W

r0
r1
r2
r3
r4
r5

Fetch
Instructions
Data
Load/Store

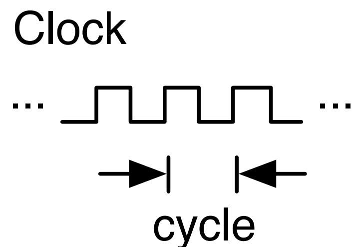
L1
(I)

L1
(D)

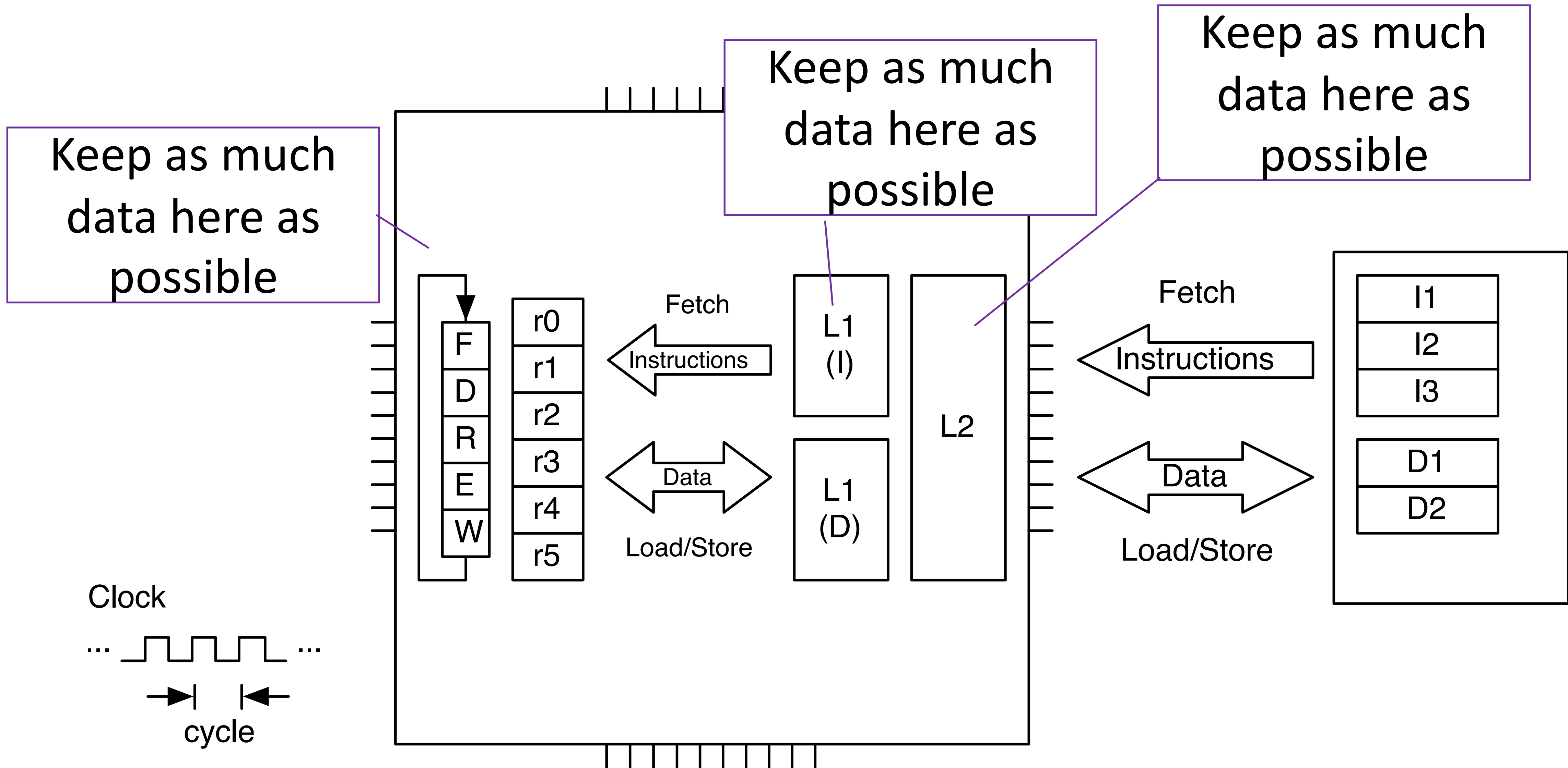
L2

Fetch
Instructions
Data

I1
I2
I3
D1
D2



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()

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

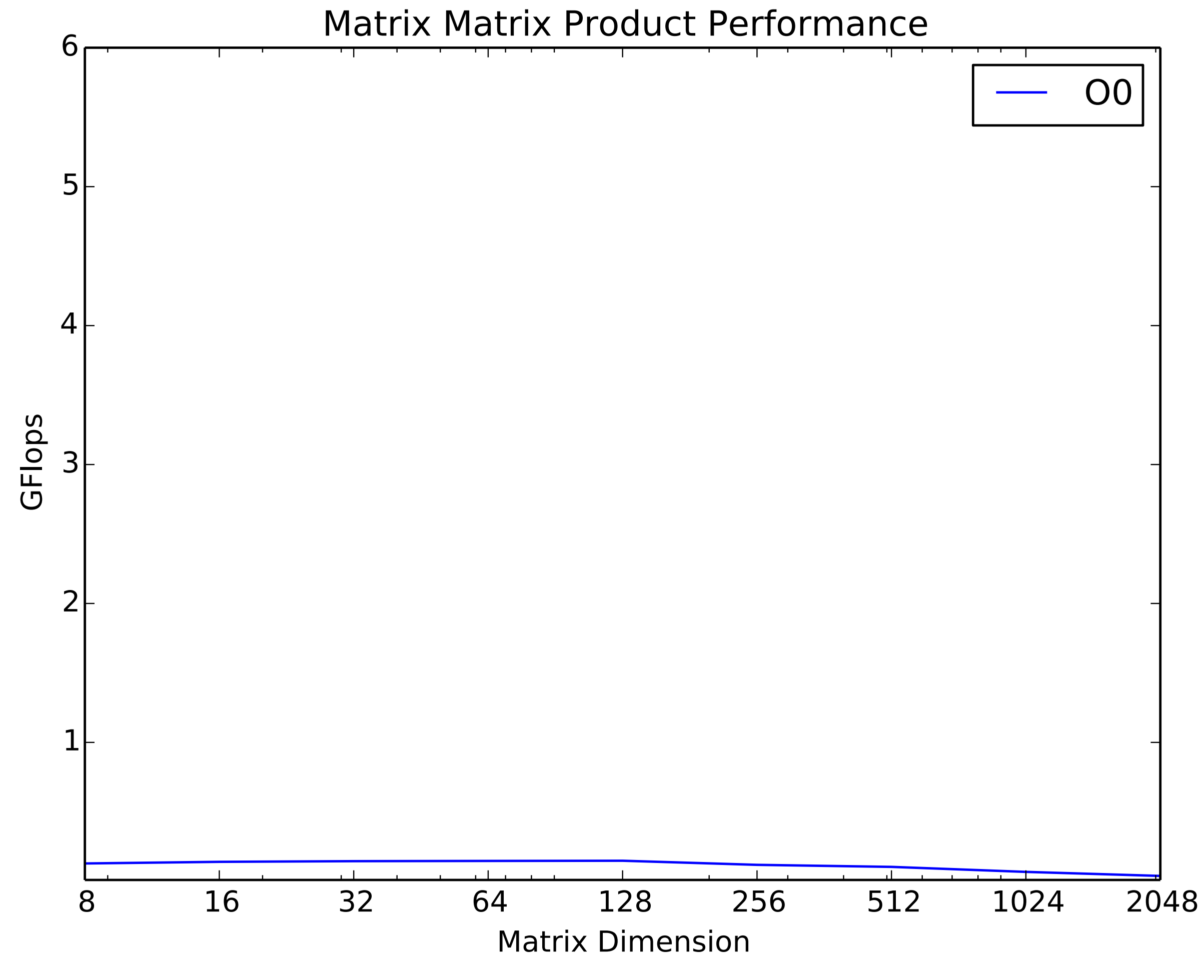
Let’s Start Benchmarking

```
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();
}
```

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

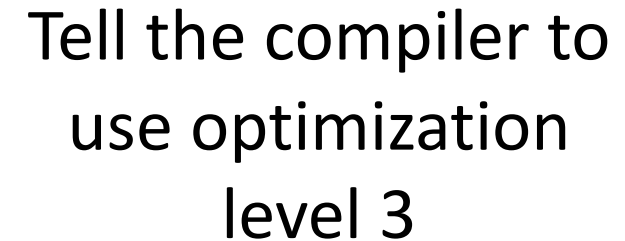
```
bench: bench.o Matrix.o
c++ -std=c++11 bench.o Matrix.o -o bench

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



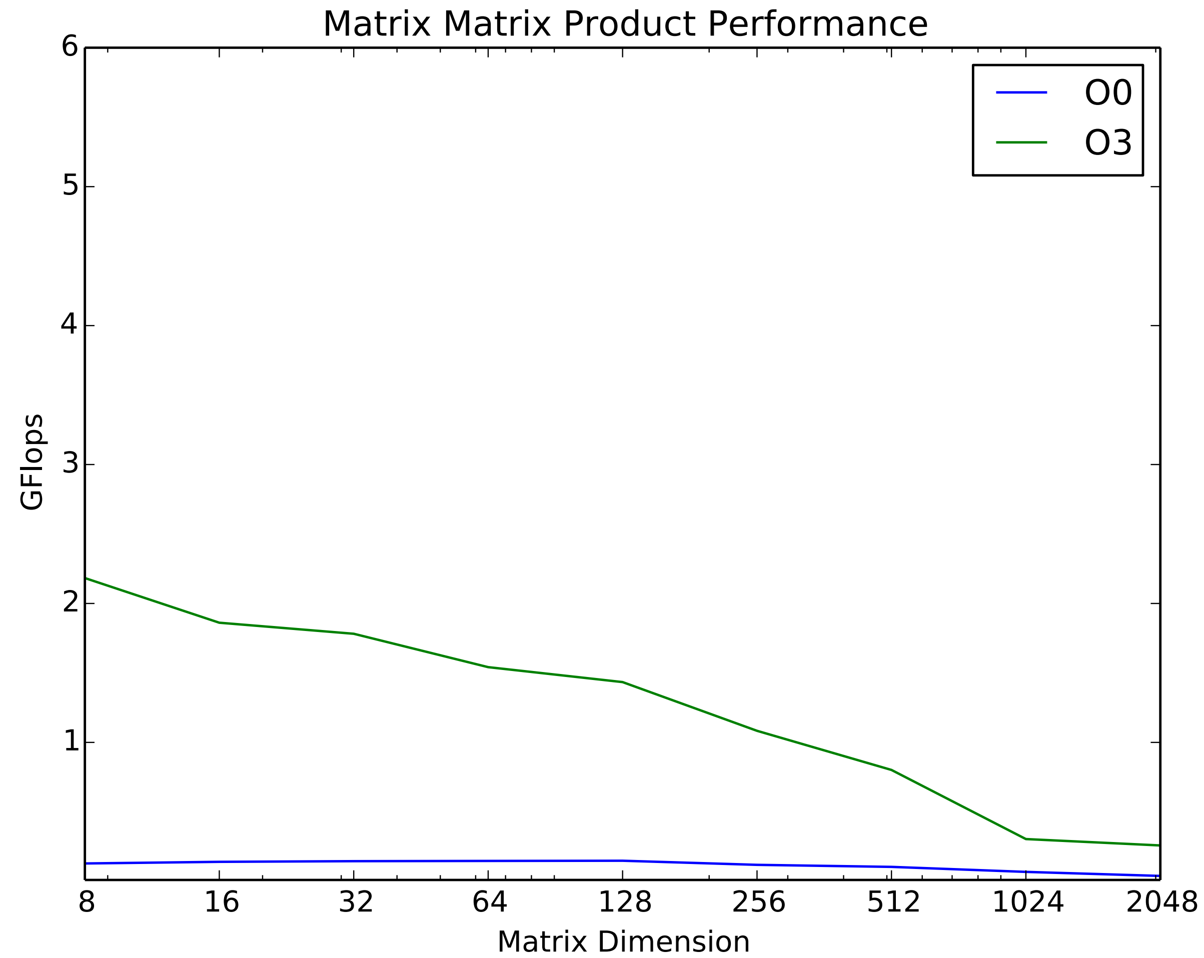
Let's Make One Small Change

```
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();
}
```




```
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

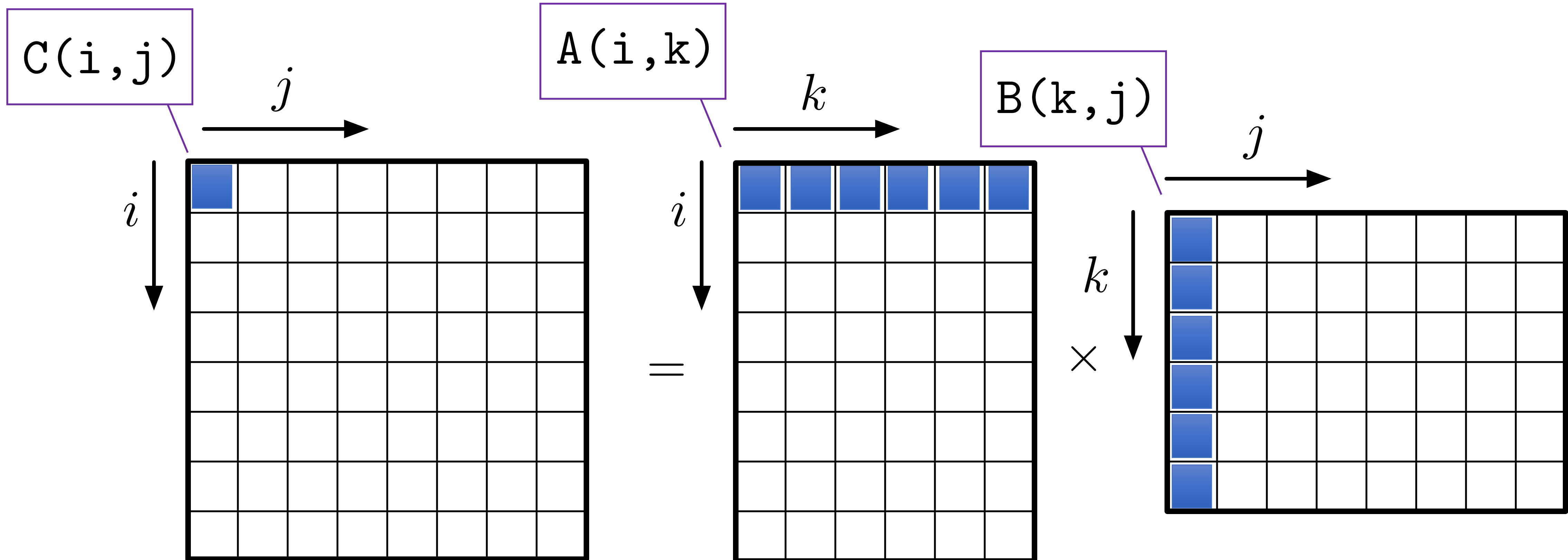
Hoist C(i,j)

```
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;
    }
  }
```

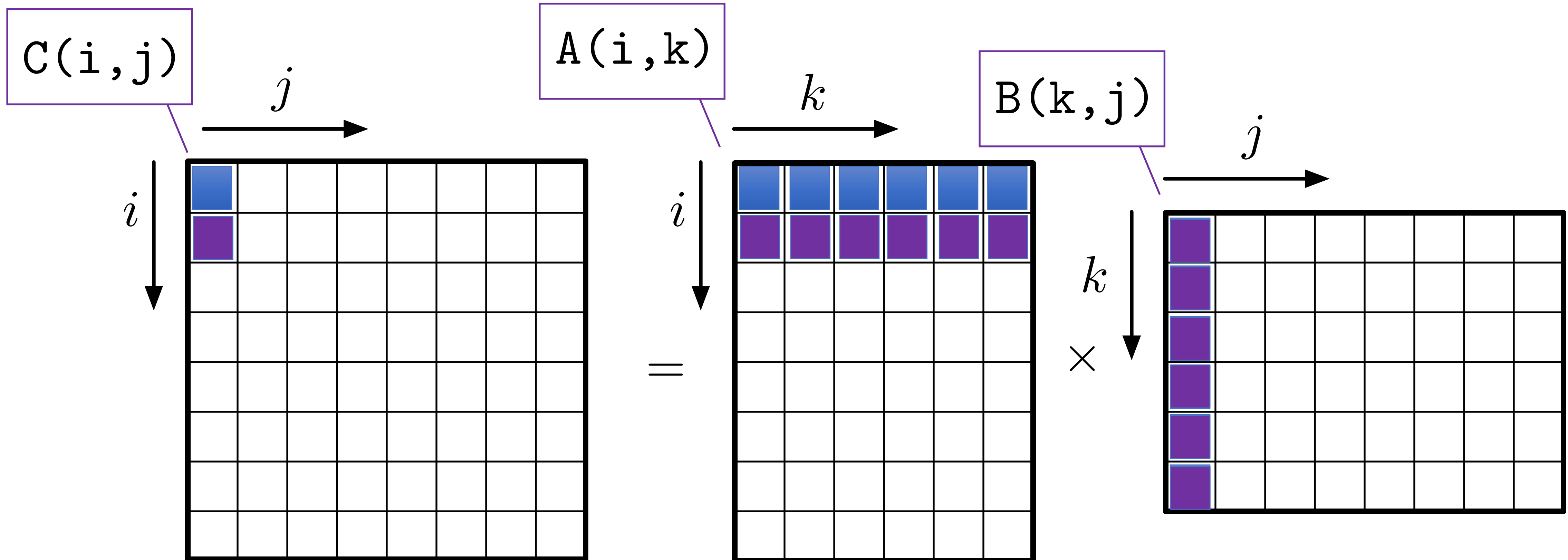
- Load `A(i,k)`
- Load `B(k,j)`
- Multiply
- Add

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

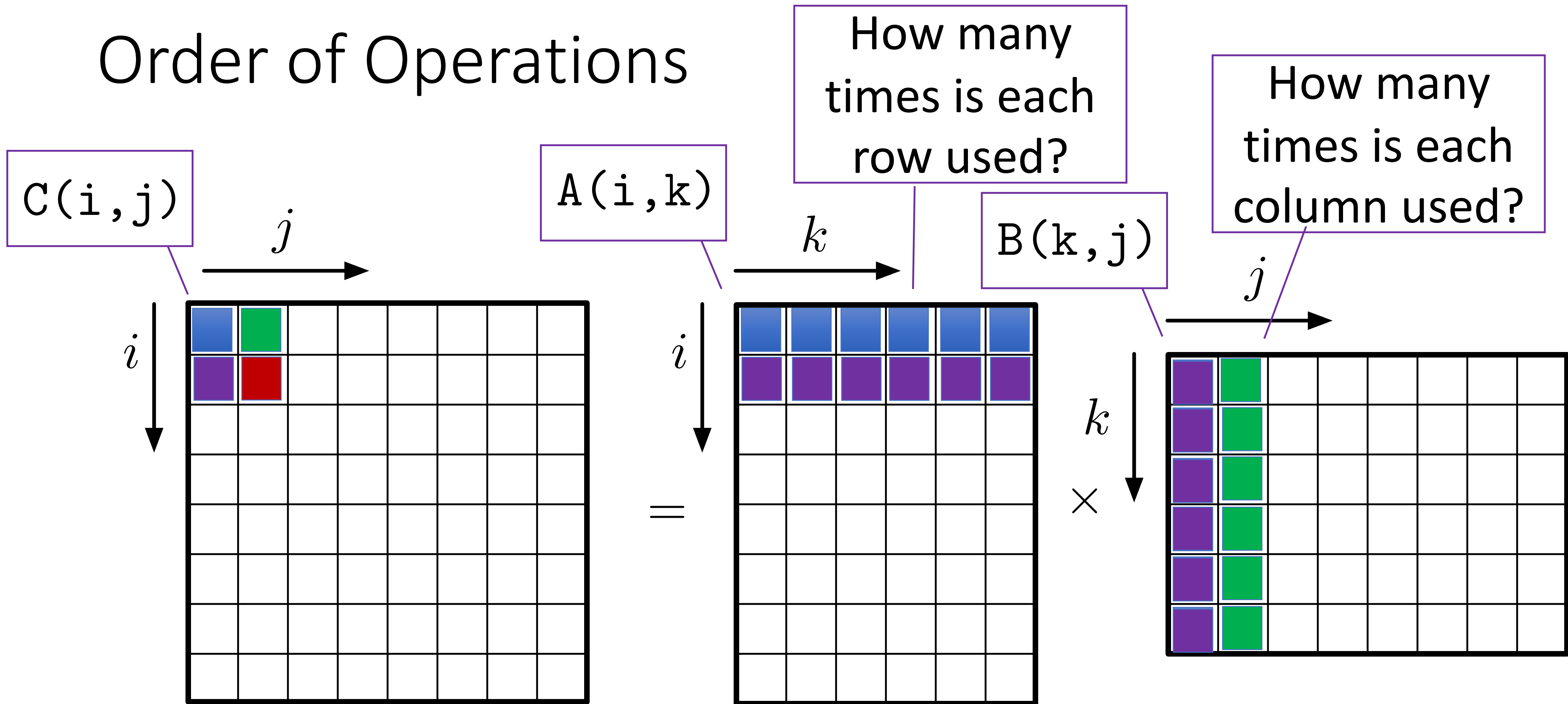
Order of Operations



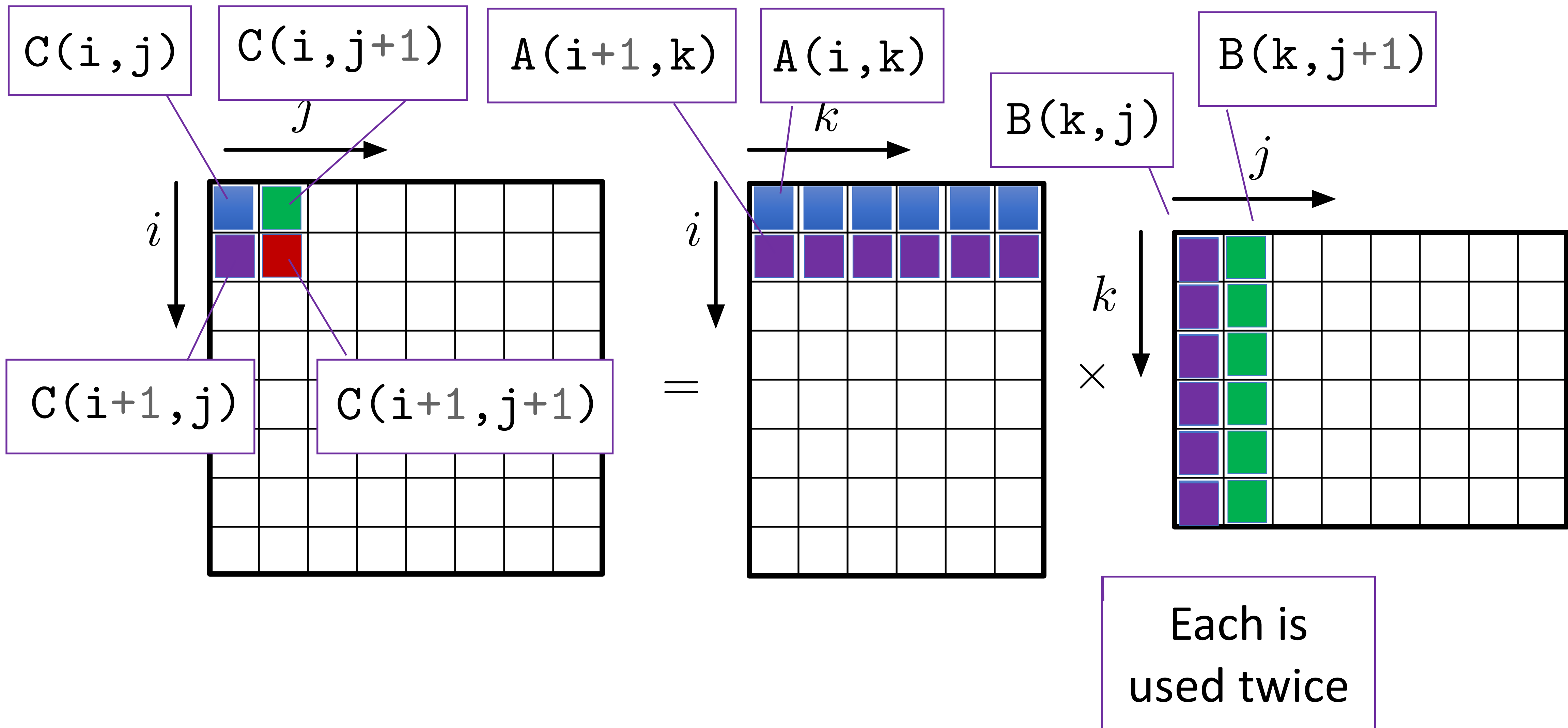
Order of Operations



Order of Operations

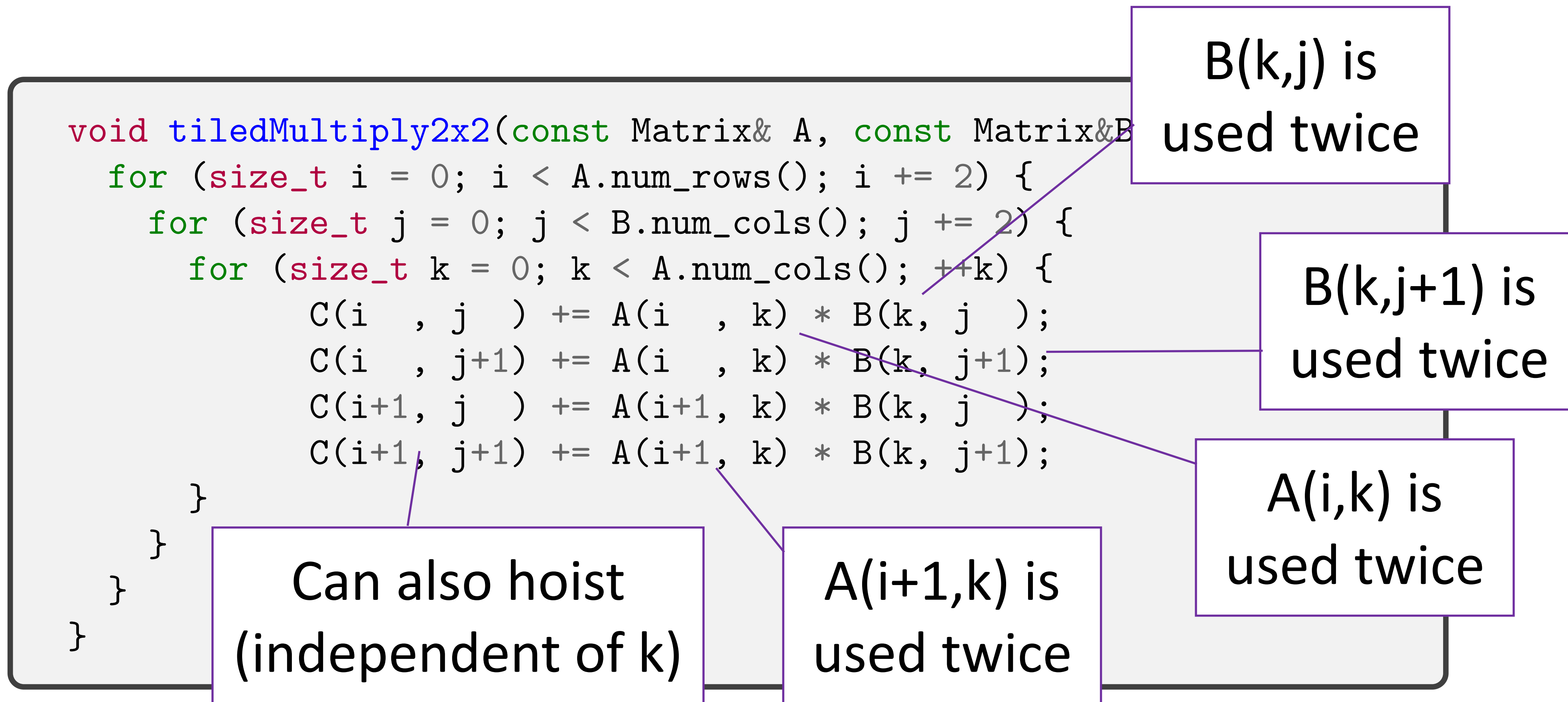


Reuse: How Many Times Are Data Reused?



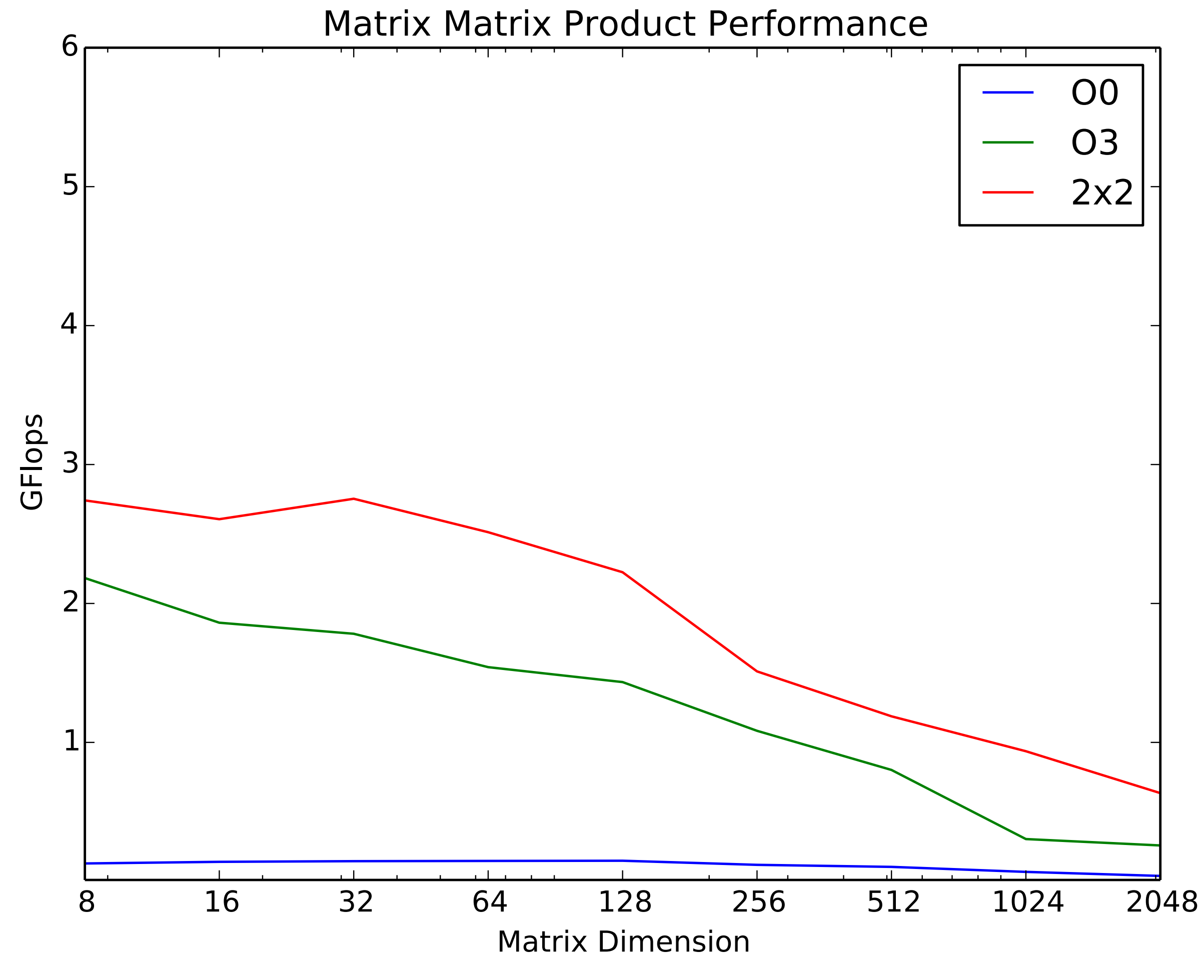
Each is used twice

Improving Locality: Unroll and Jam

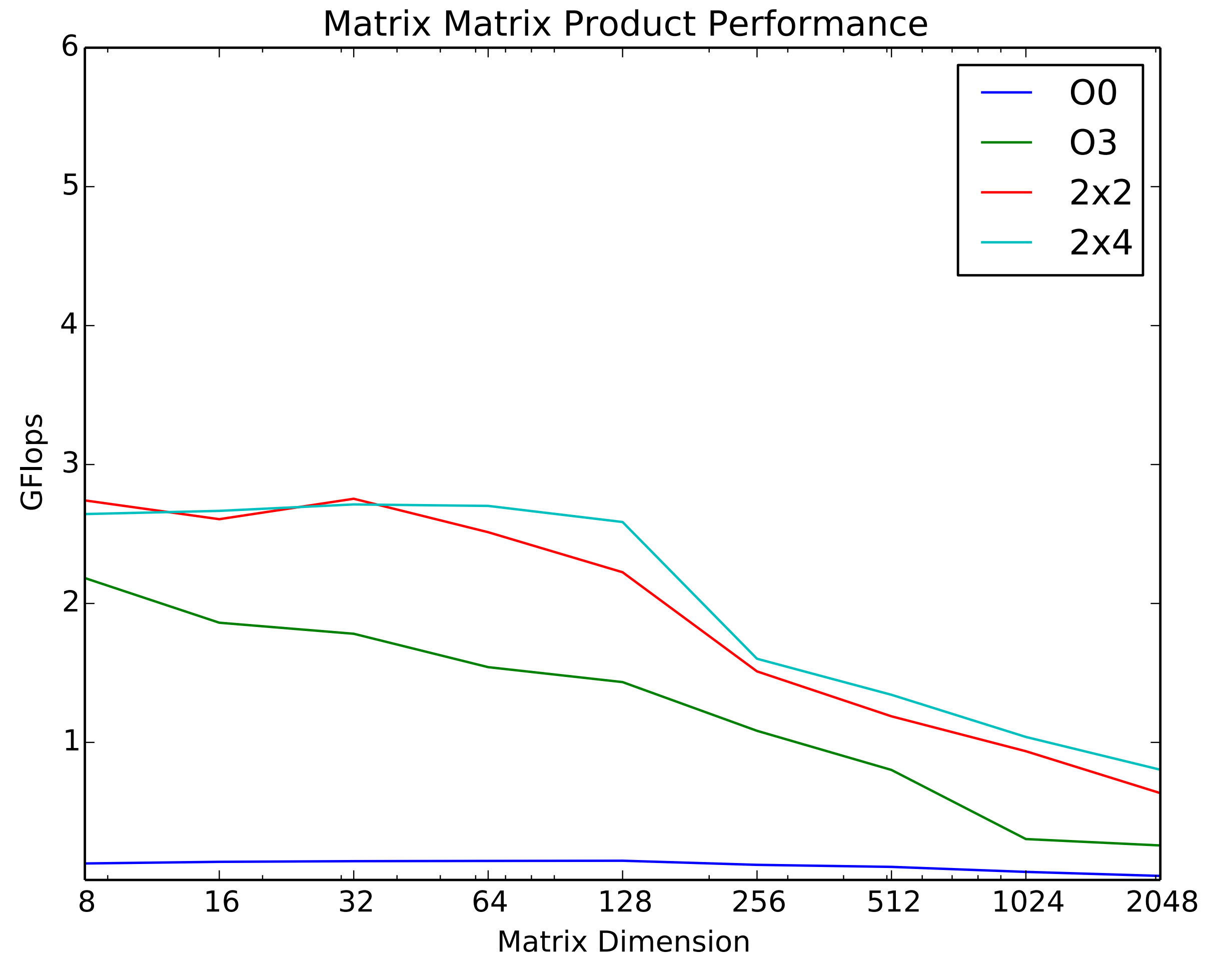


- Four memory operations and eight floating point operations per iteration
- 8/12 = 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

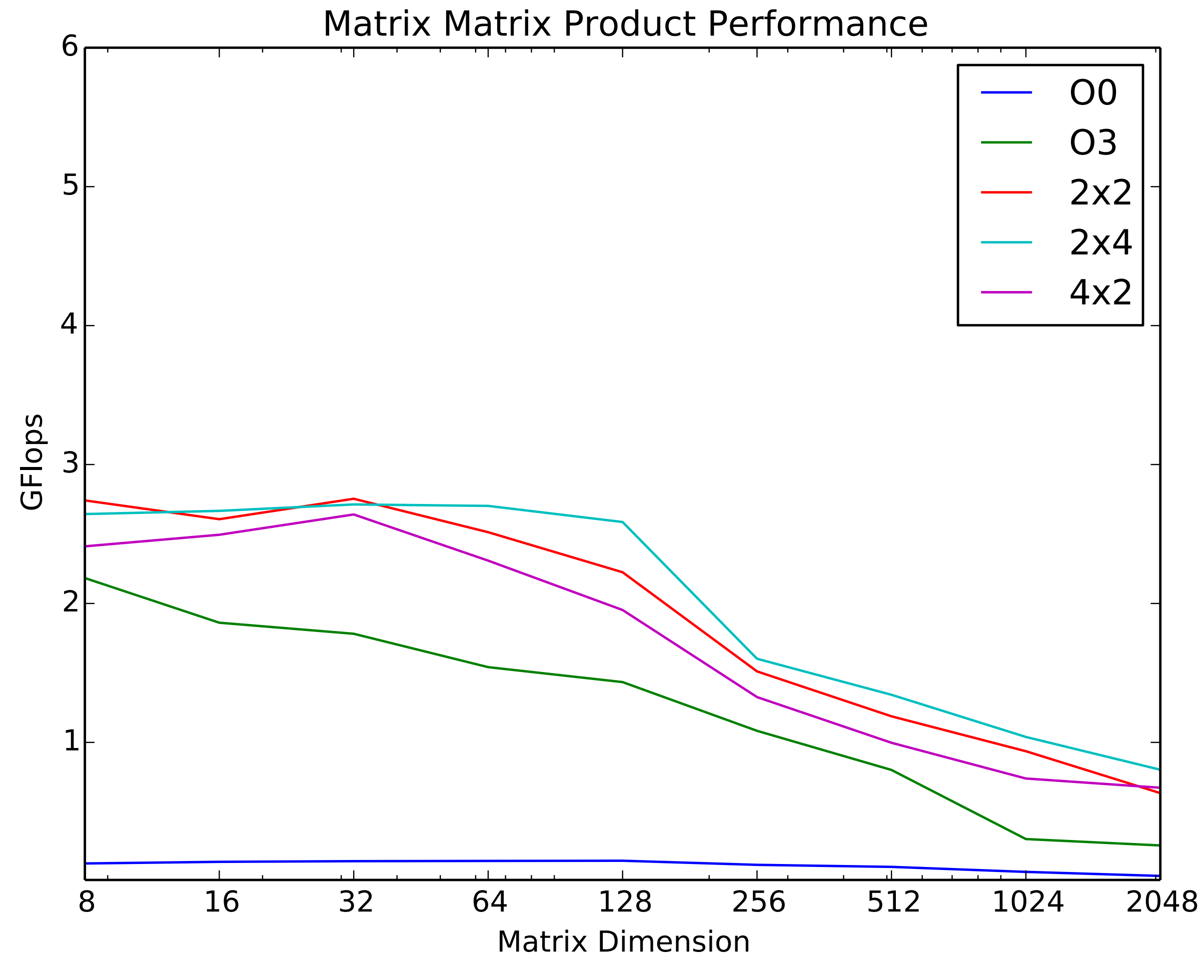
Example: Register Locality



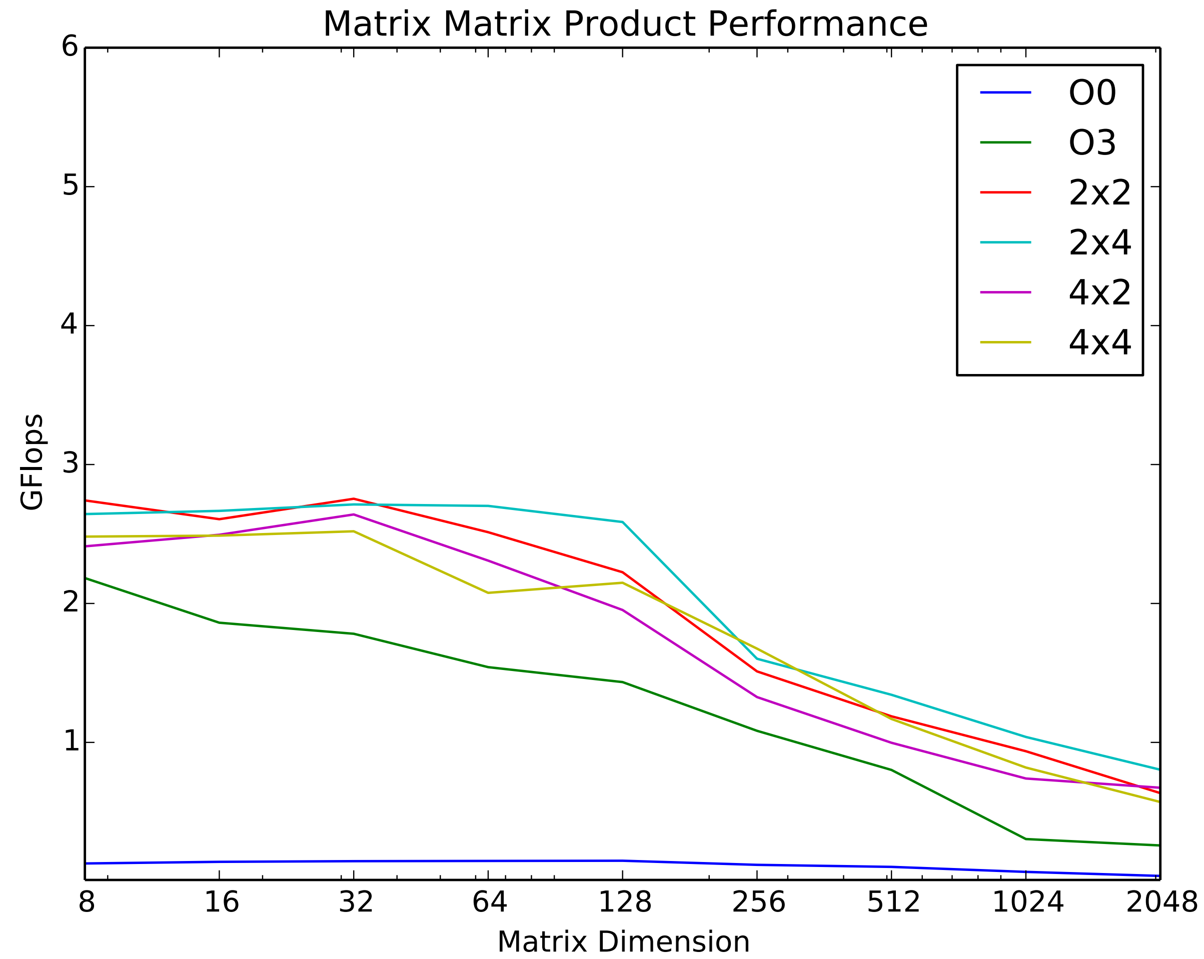
2 by 4



4 by 2



4 by 4

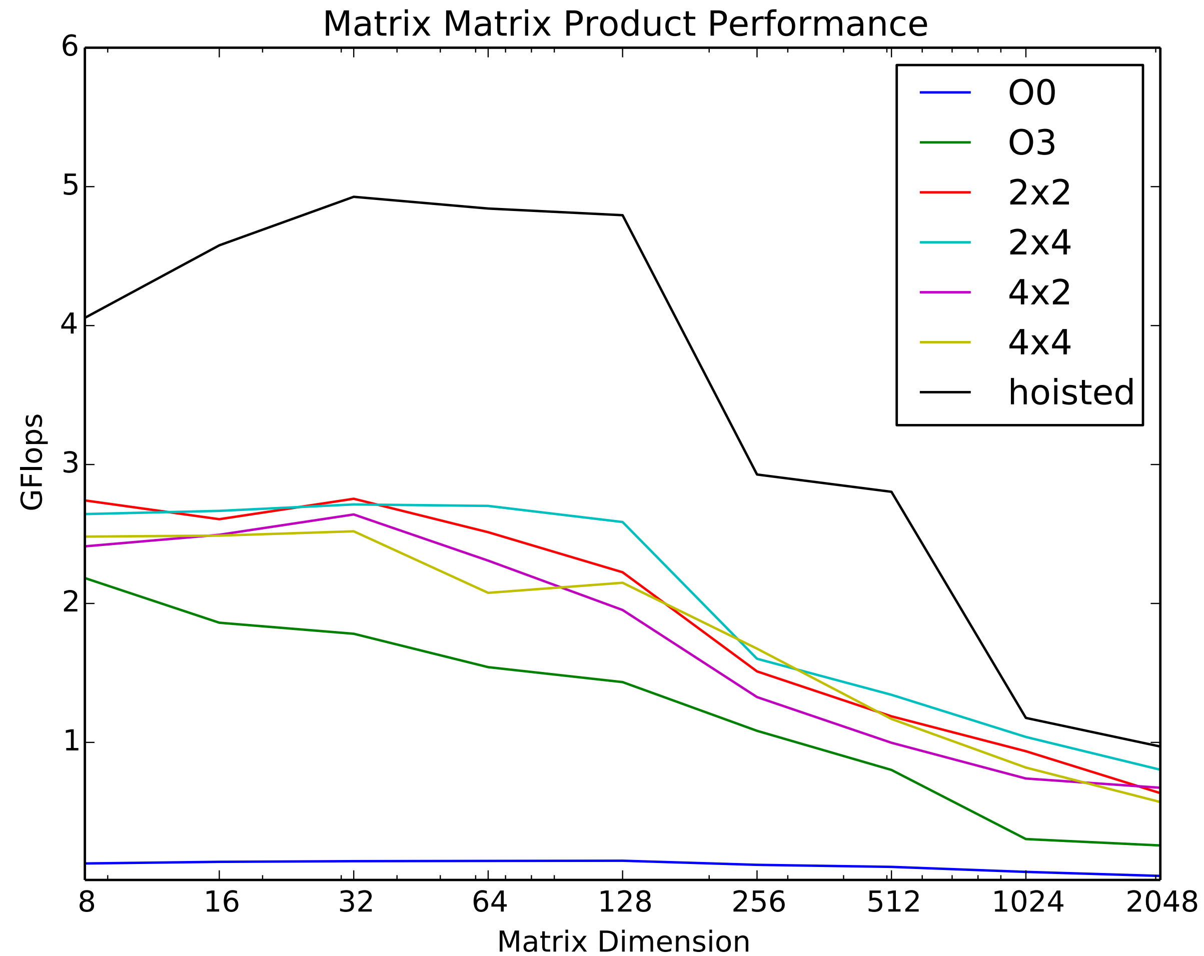


Unrolling and Hoisting

```
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

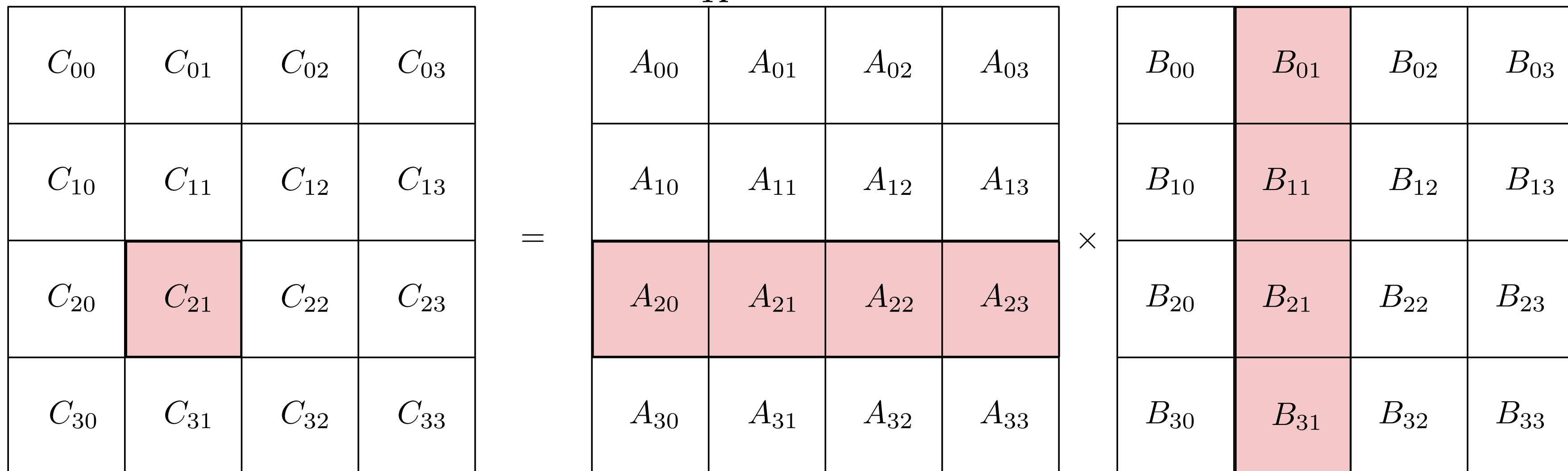
Unrolling and Hoisting



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

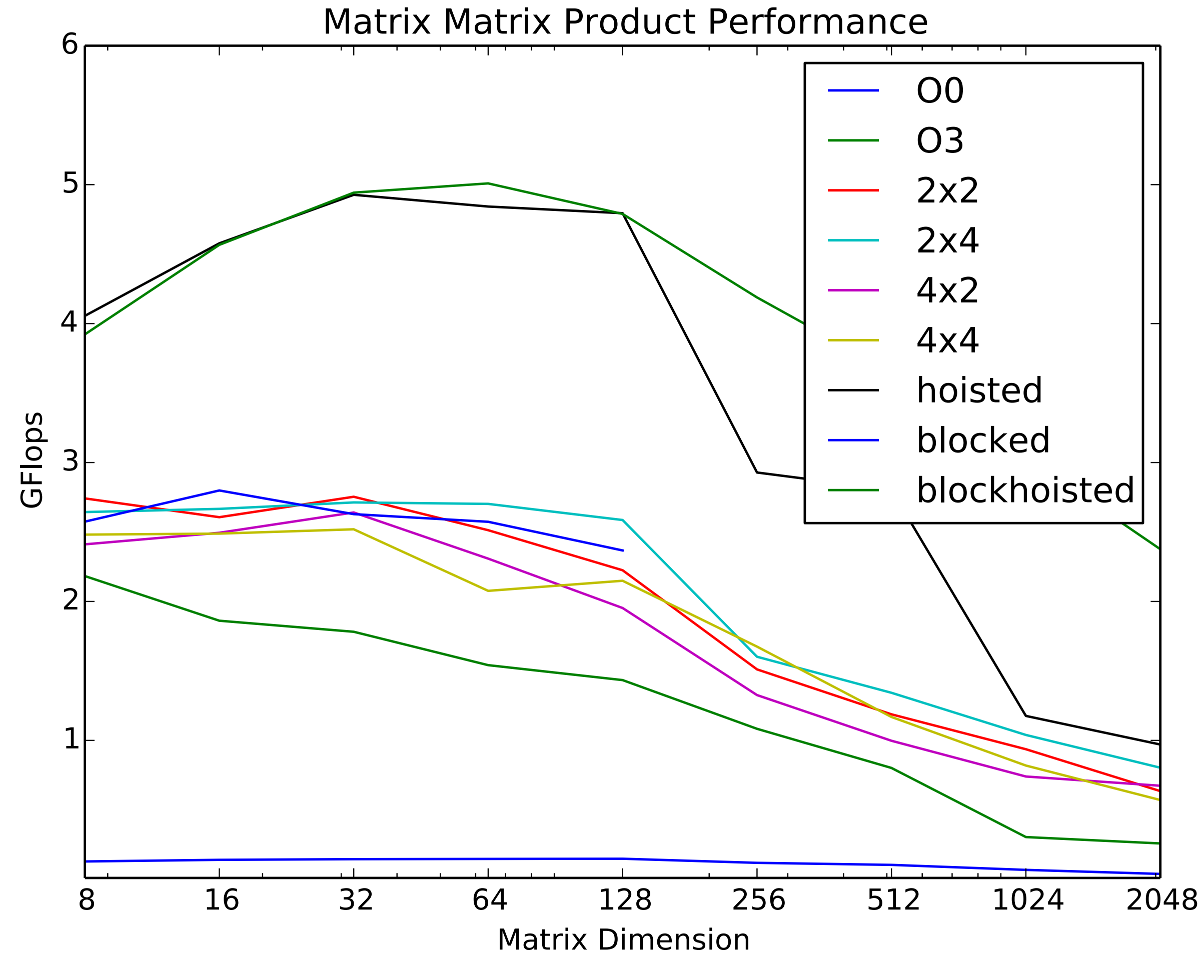
Blocking and Unrolling

```
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);
            }
          }
        }
      }
    }
  }
}
```

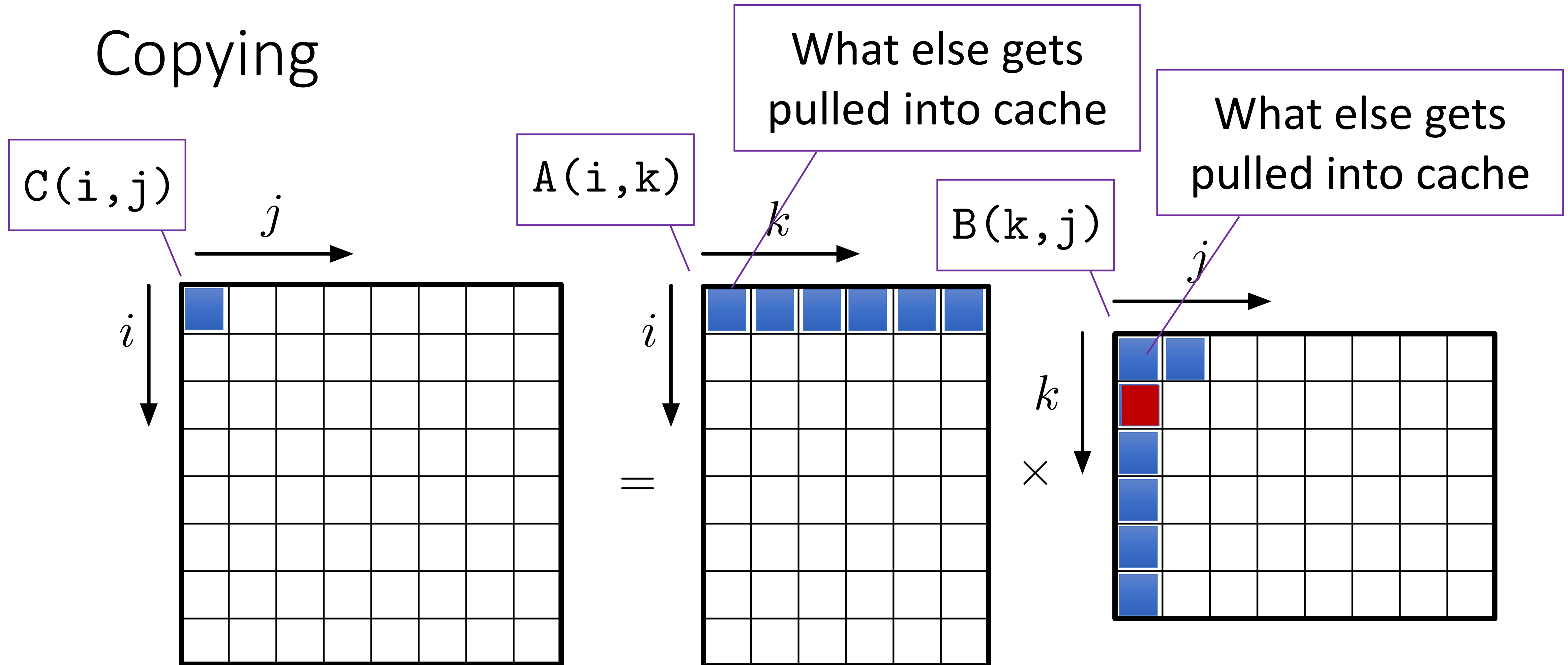
Outer loops work across blocks (for each block)

Inner loops work on blocks

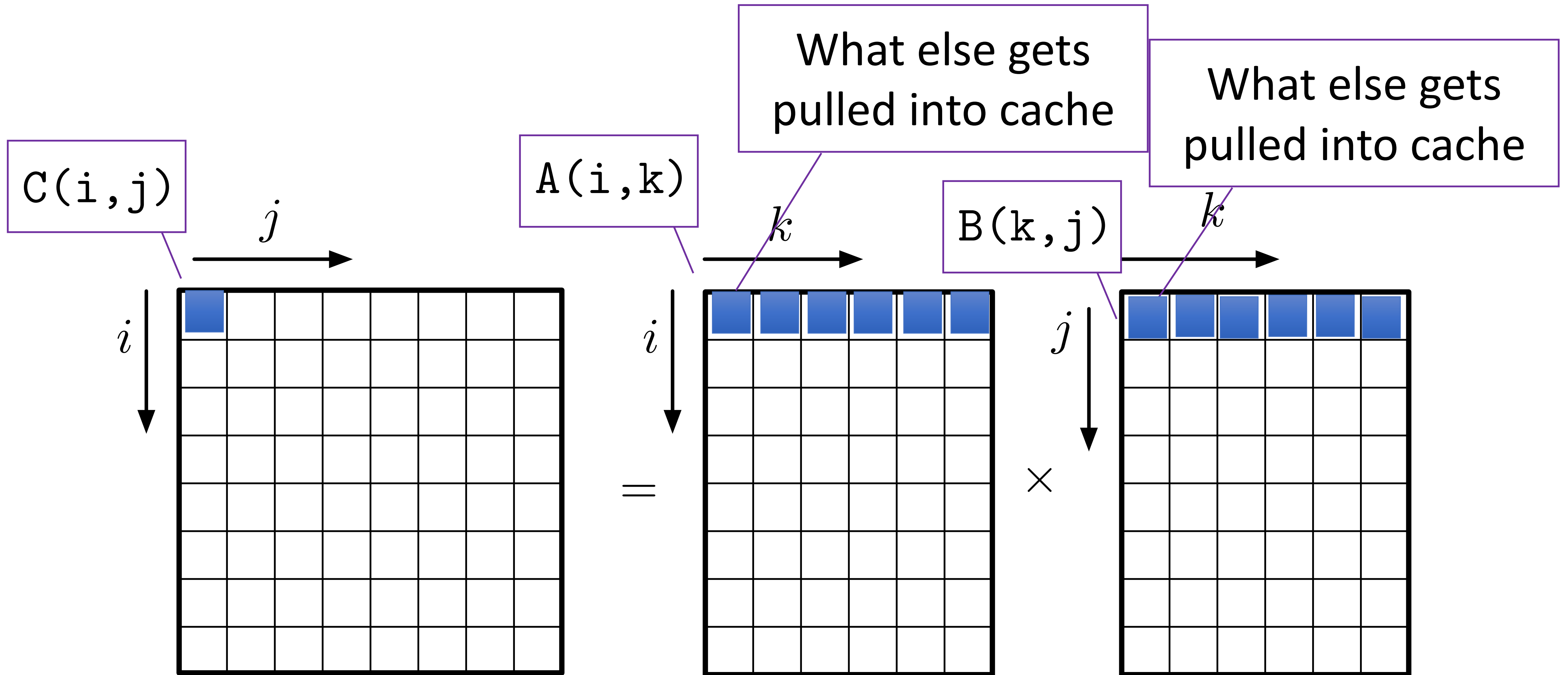
Blocking and Unrolling and Hoisting



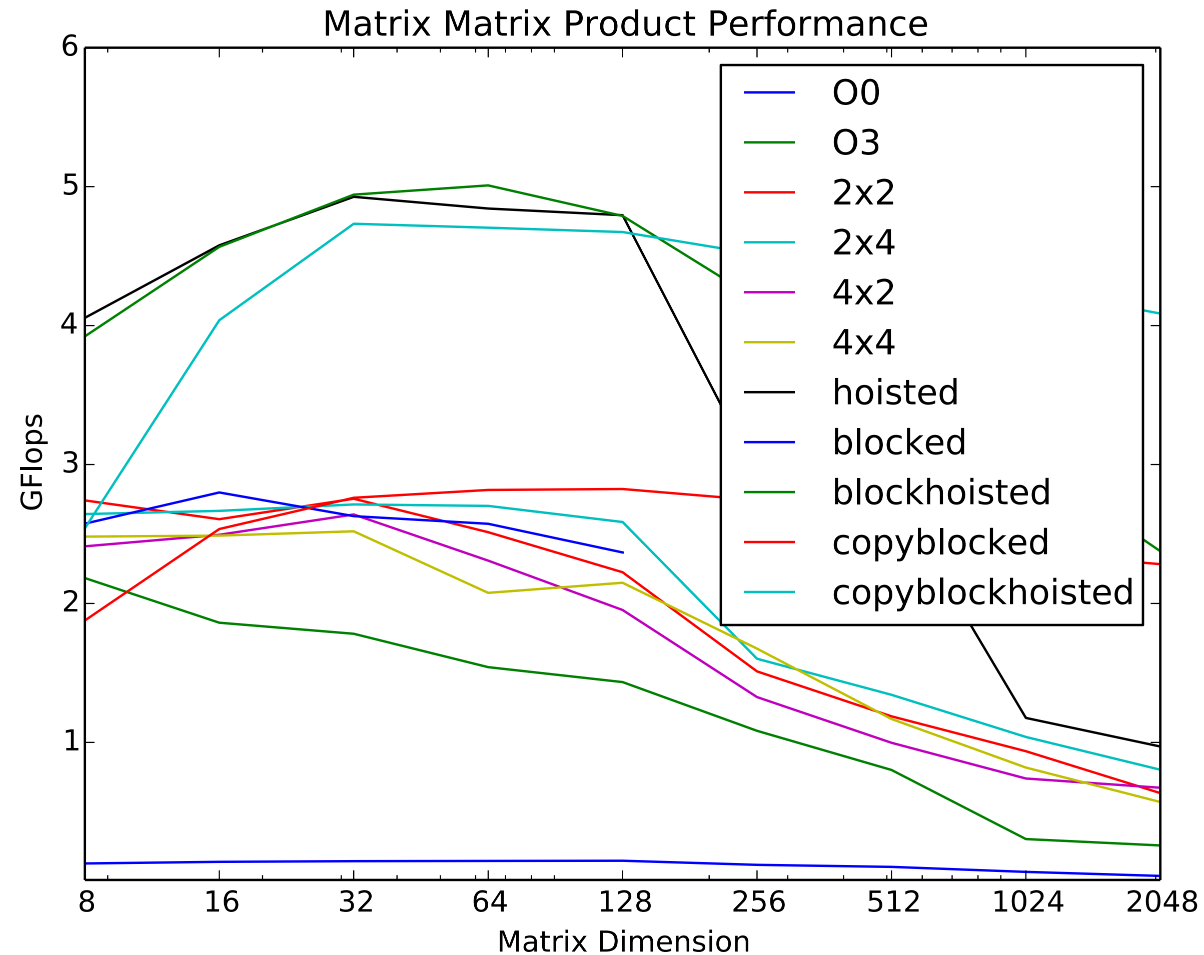
Copying



Copying and Transpose Multiply



Blocking and Unrolling 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 di 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

- (cf) also "Goto" BLAS and FLAME (Goto, van de Geijn)

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  );
            }
          }
        }
      }
    }
  }
}
```

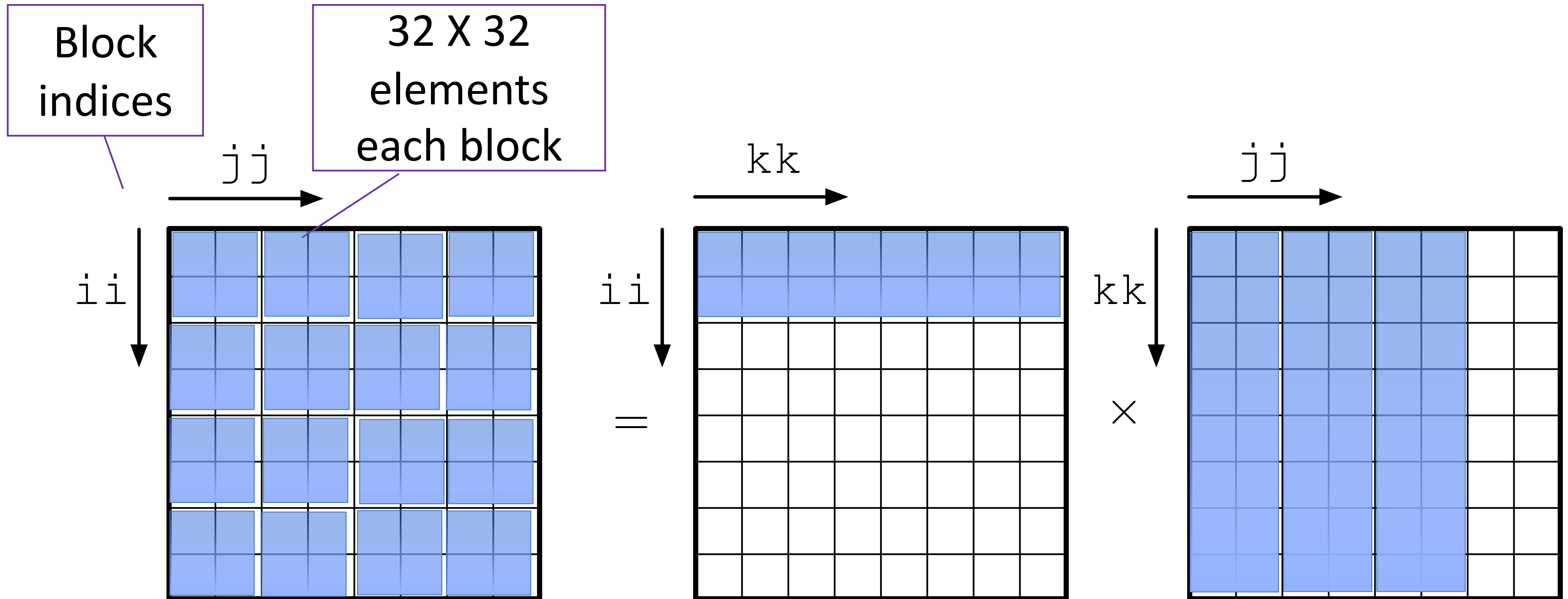
For illustrative purposes only

Example code (slides and source) omit this

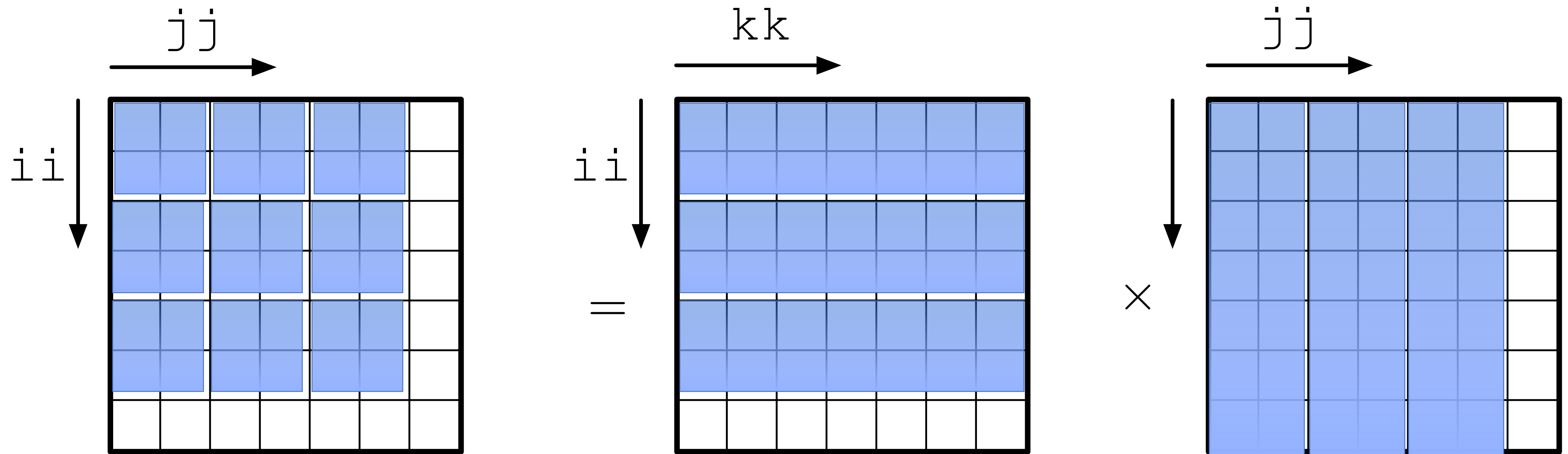
Caveat codor

Powers of 2 are great

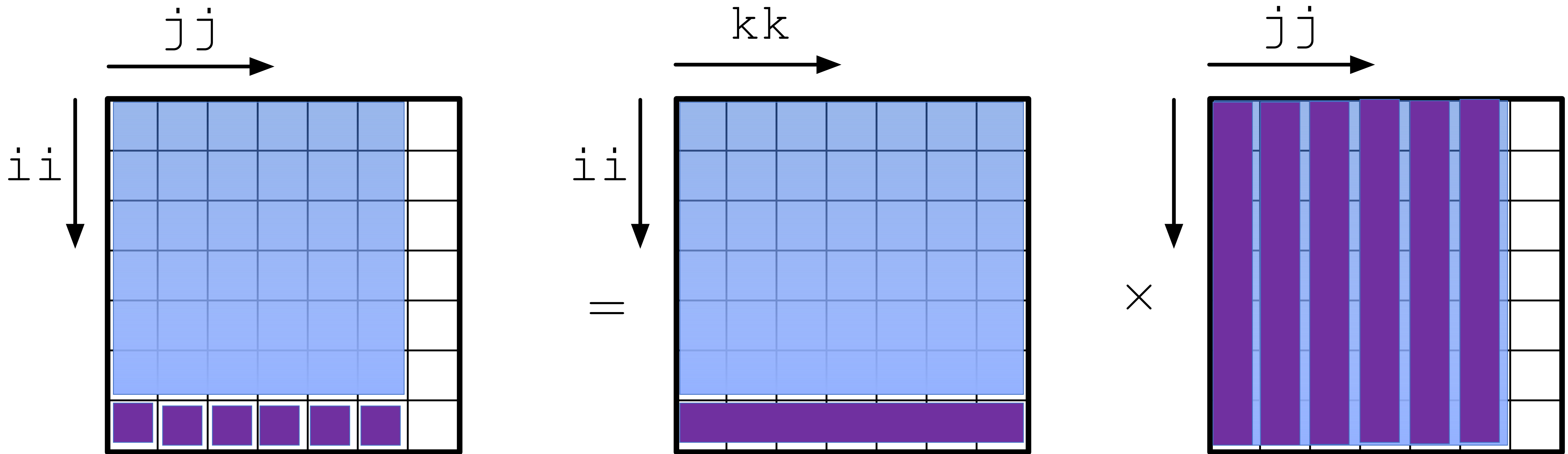
Cleanup (Fringe)



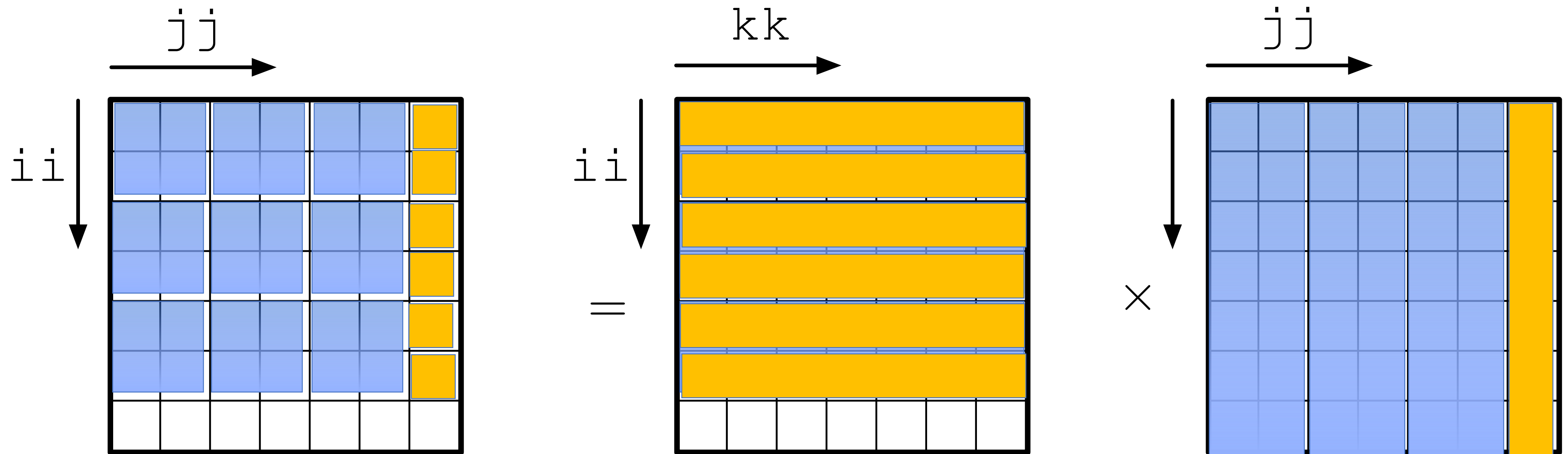
Fringe Cleanup



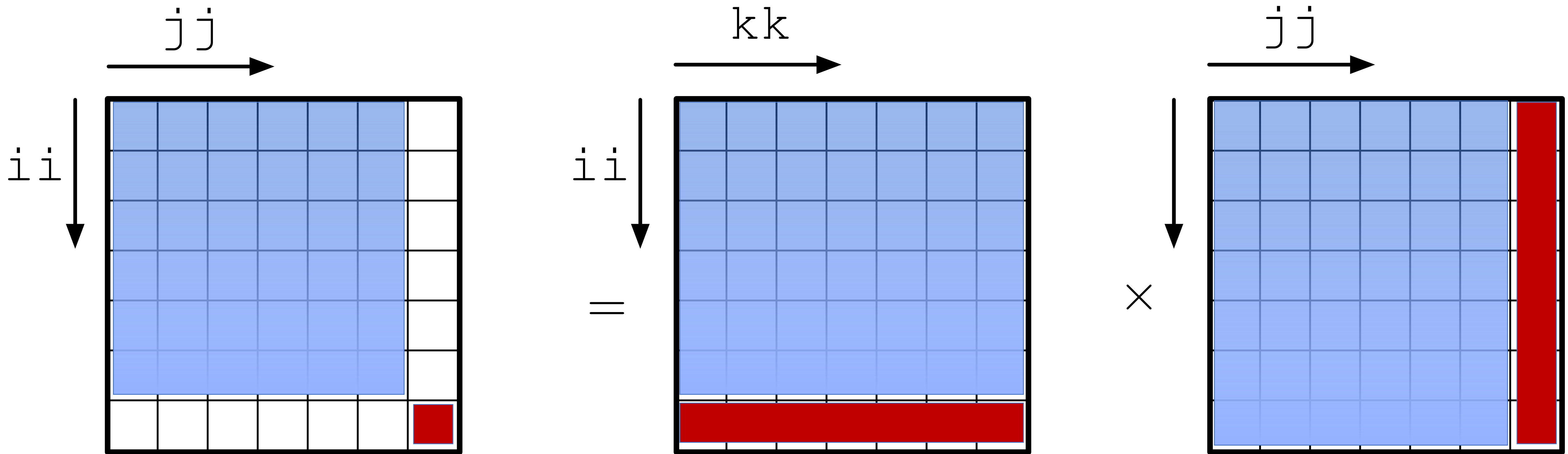
Fringe Cleanup



Fringe Cleanup



Fringe Cleanup



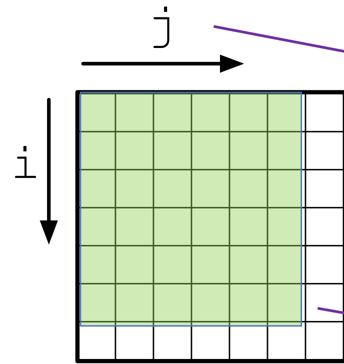
Fringe Cleanup

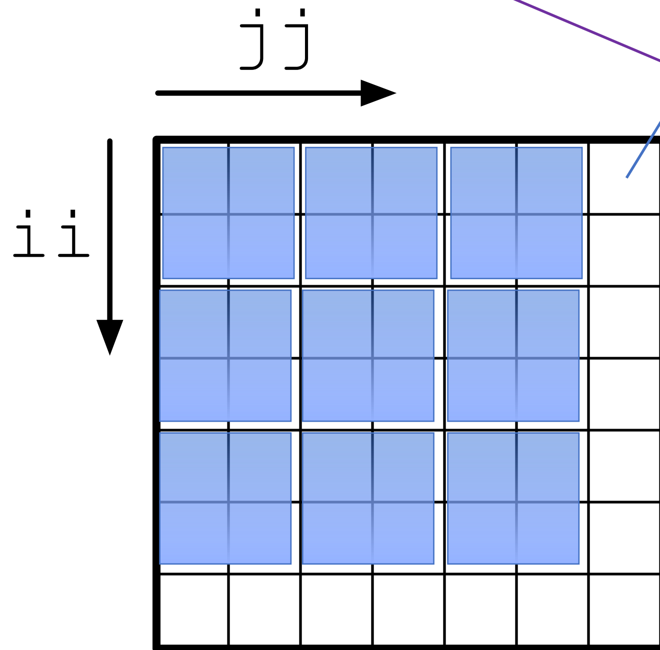
Same problem within each block

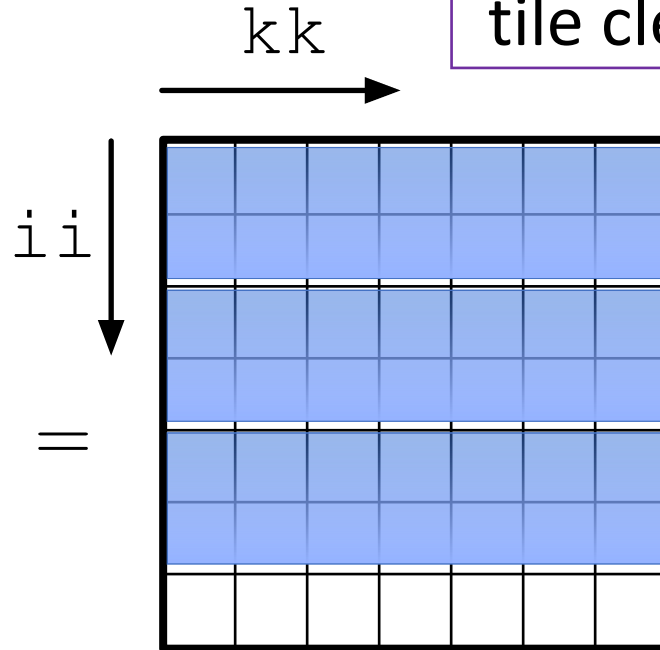
i and j count by tile size

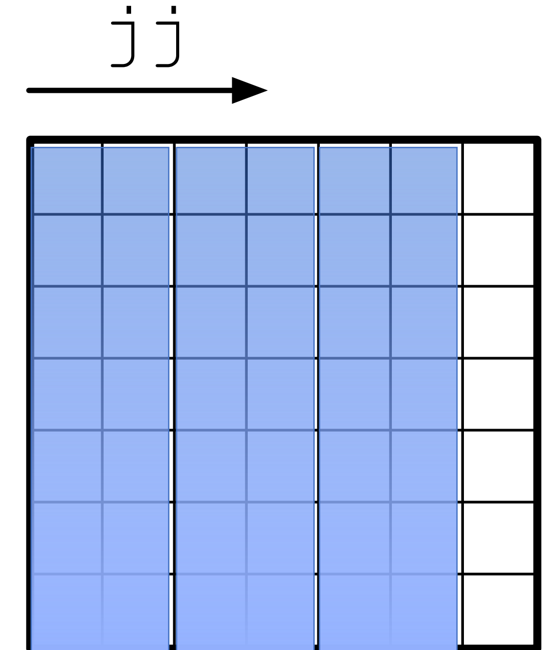
Block fringe might not be divisible by tile size

Also need tile cleanup







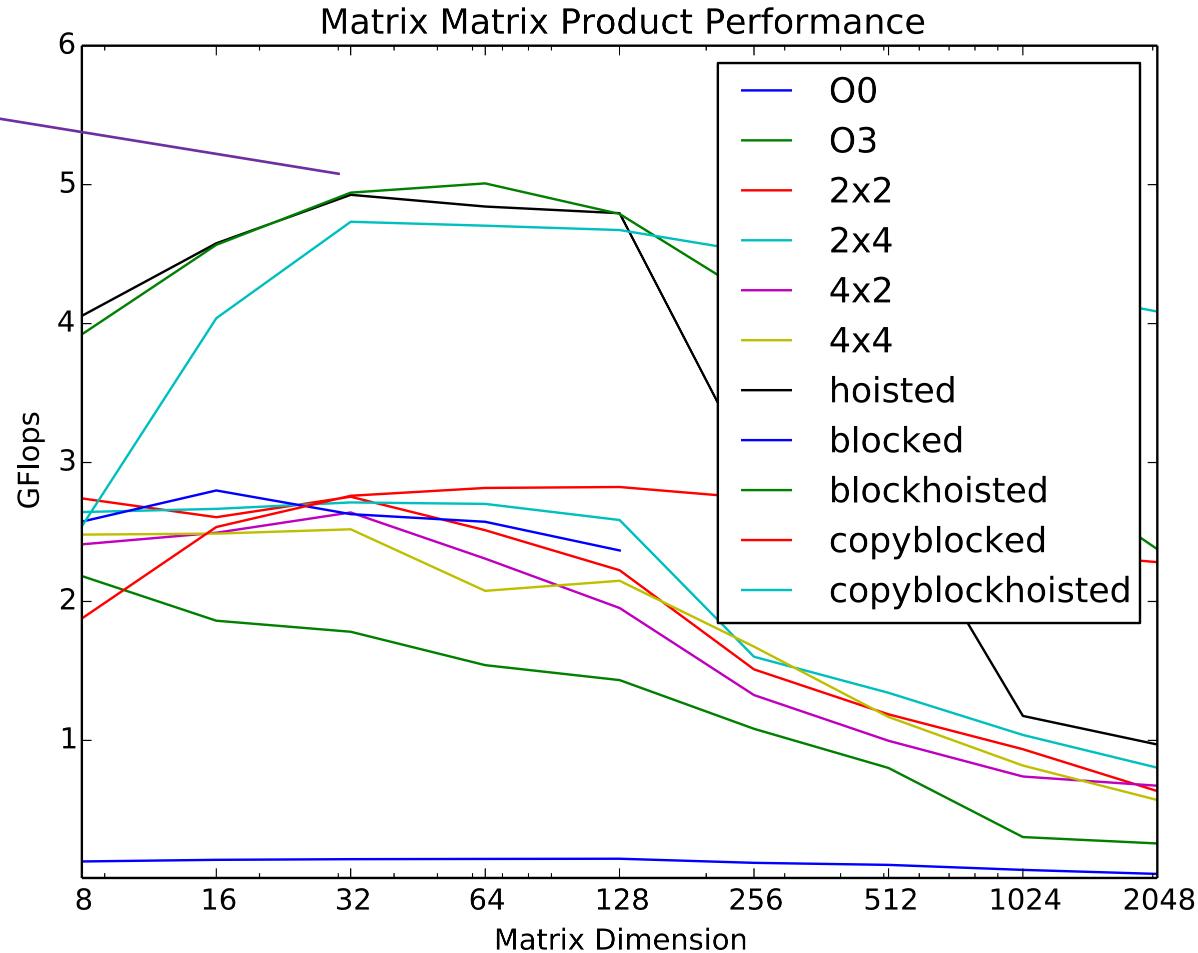


Blocking and Unrolling and Hoisting and Copying

Is this the best we can do?

How good is it anyway?

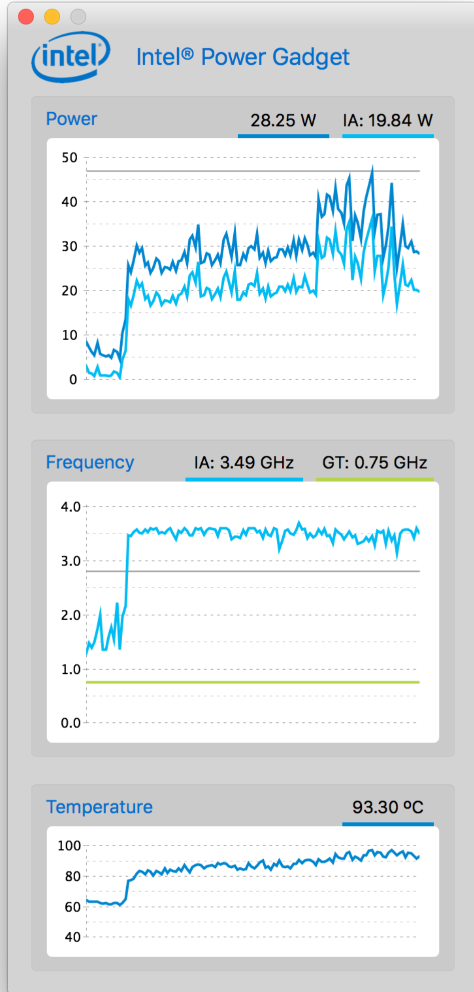
(cf PS 4)

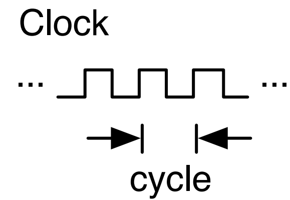


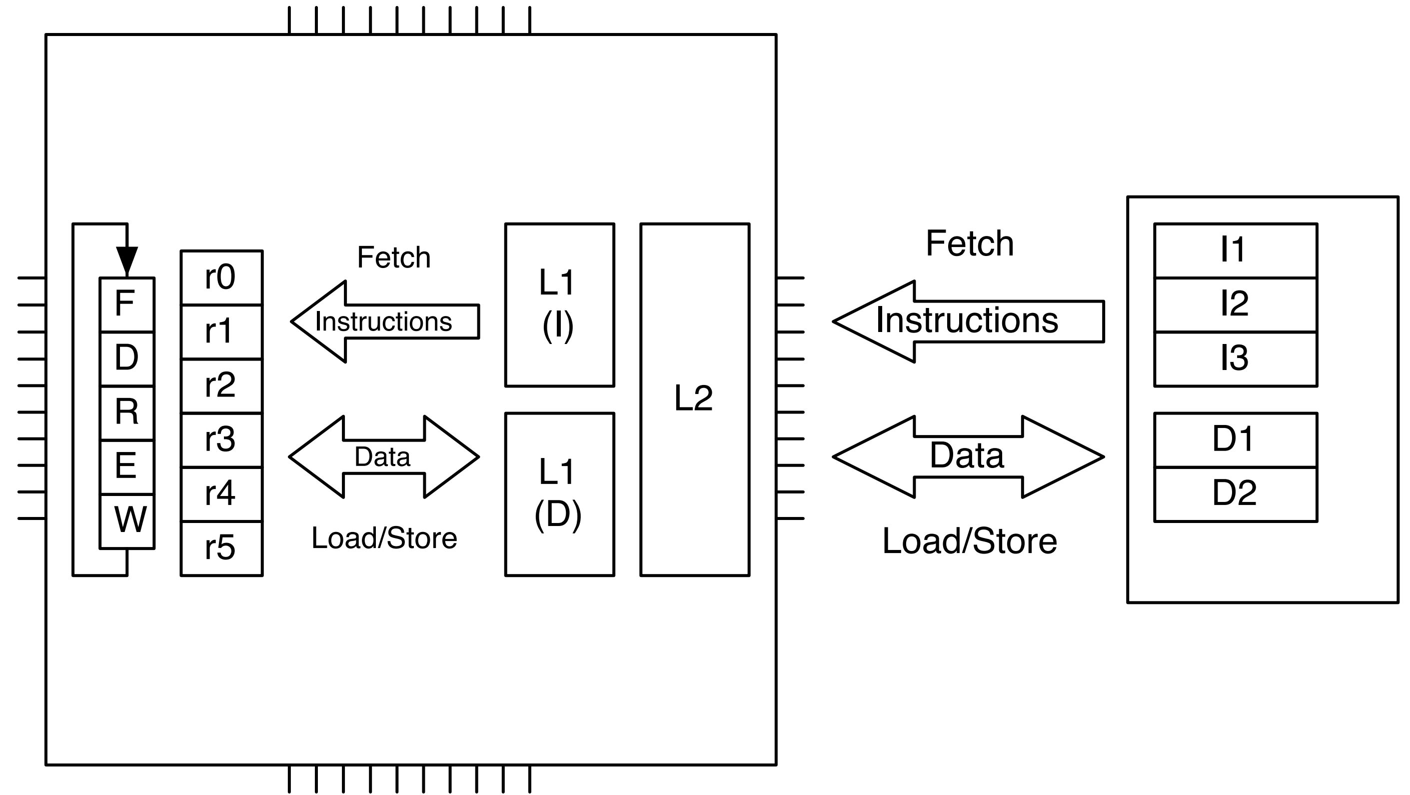
Calypso

- Program for generating matrix-matrix products

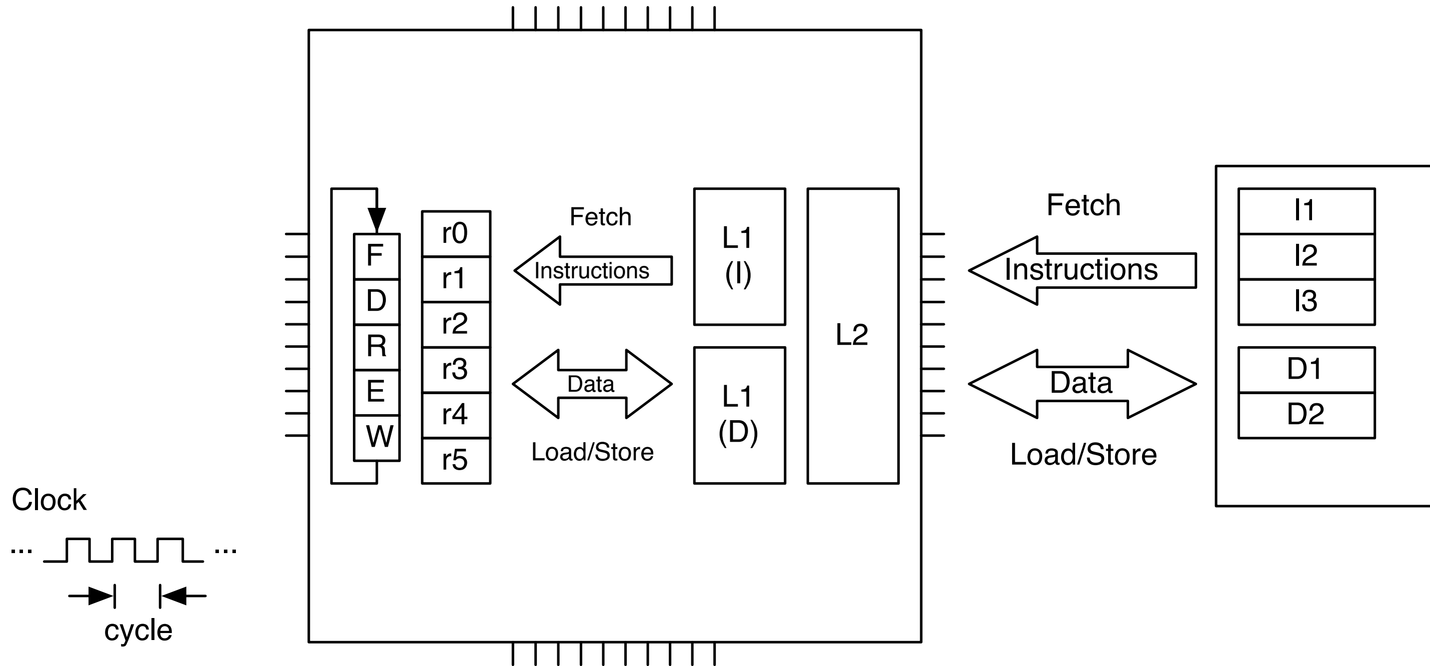
CPU Clock Speed

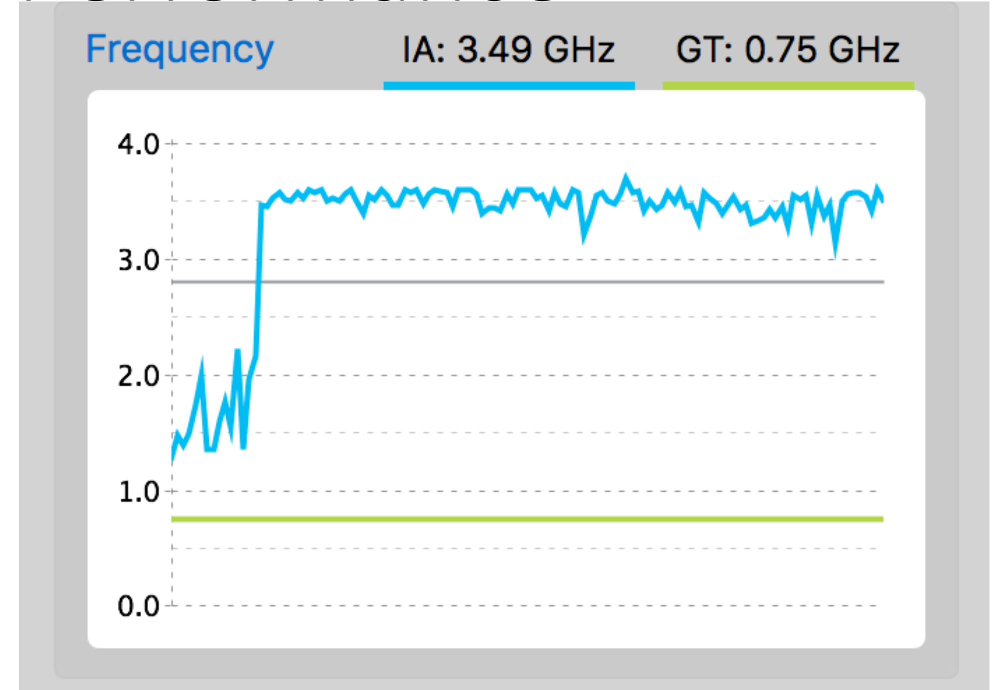






Peak Performance vs Achieved Performance





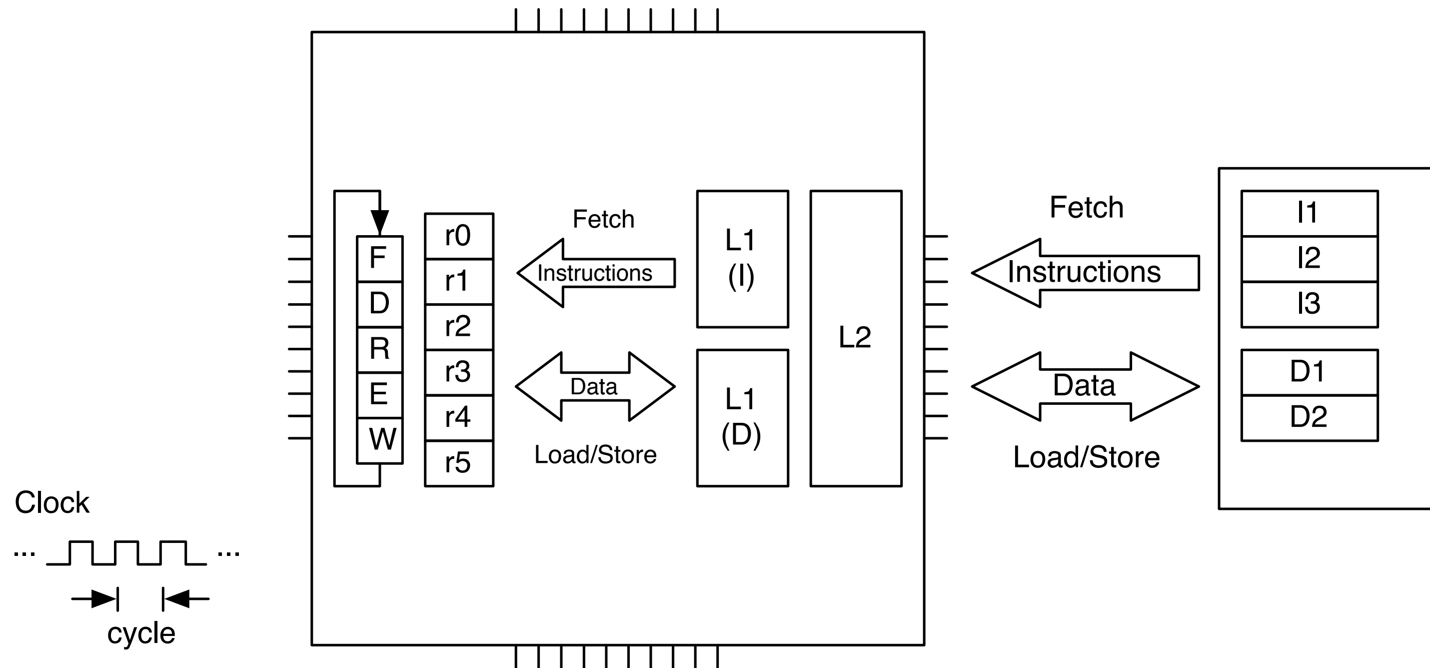
$$5 \times 10^9 \frac{\text{FLOPS}}{\text{second}} \div 3.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \approx 1.5 \frac{\text{FLOPS}}{\text{cycle}}$$

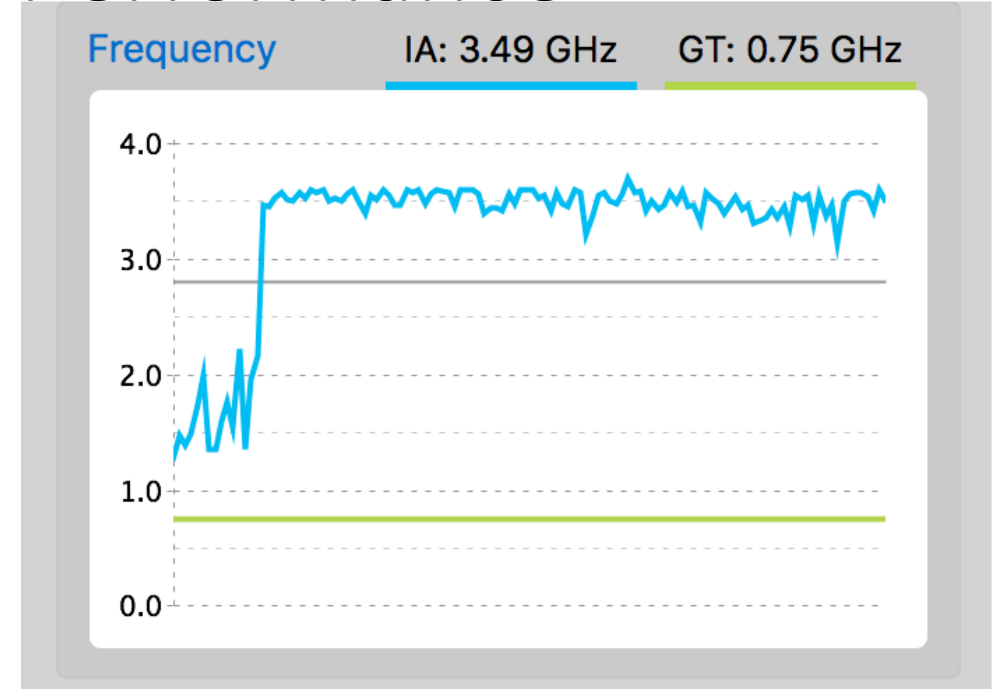
5 GFLOPS

3.5 GHz

Does this make sense?

Peak Performance vs Achieved Performance



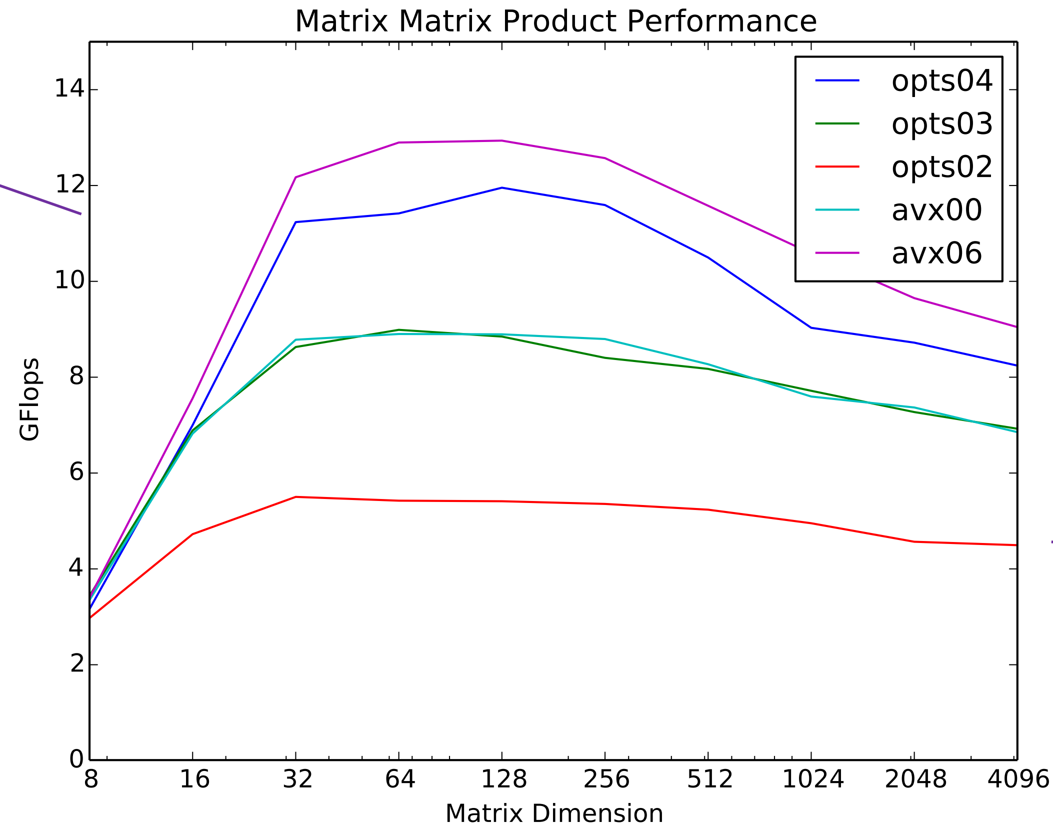


$$5 \times 10^9 \frac{\text{FLOPS}}{\text{second}} \div 3.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \approx 1.5 \frac{\text{FLOPS}}{\text{cycle}}$$

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 (AVX)

(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

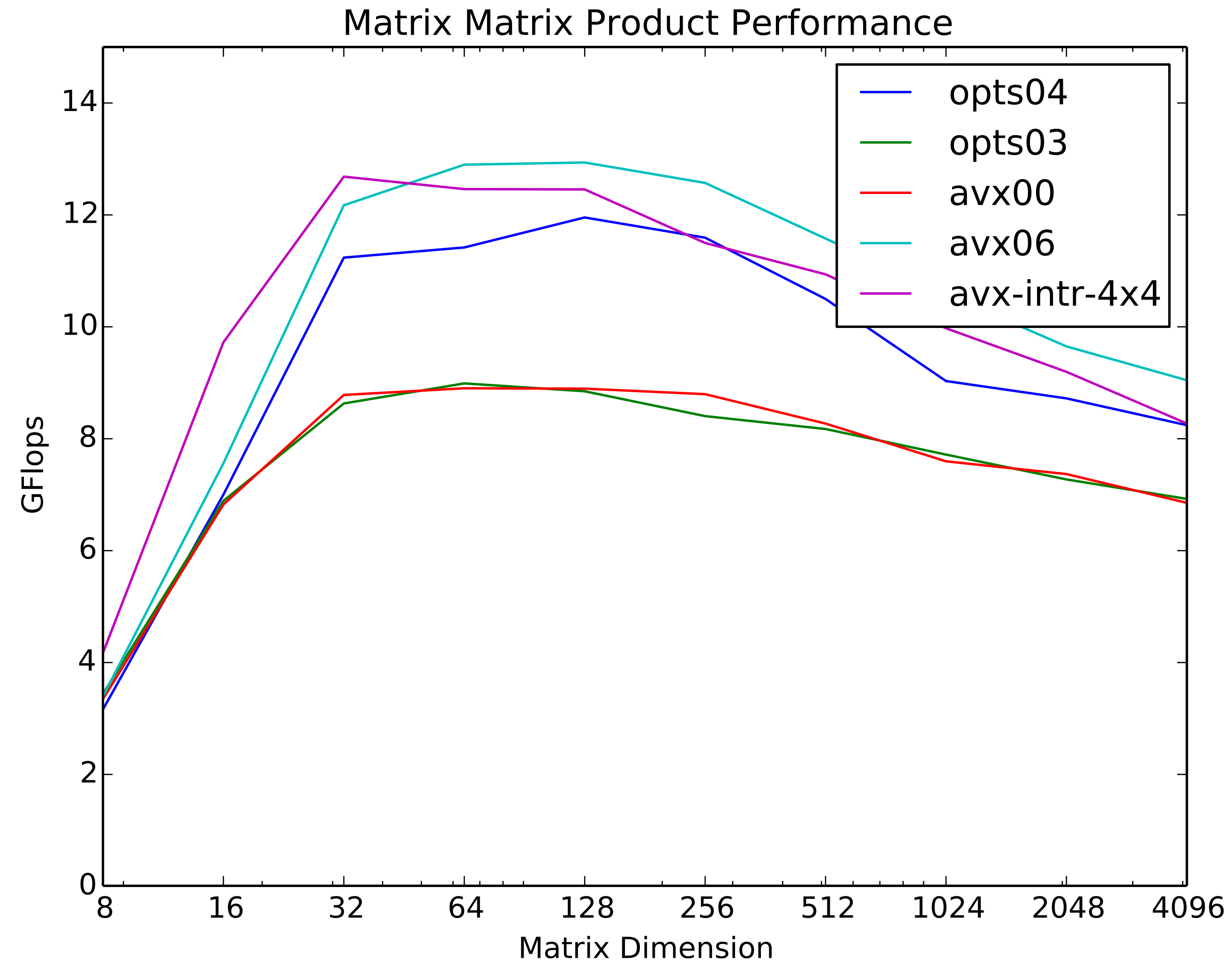
Four doubles

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;
  }
}
```

Fused Multiply-Add

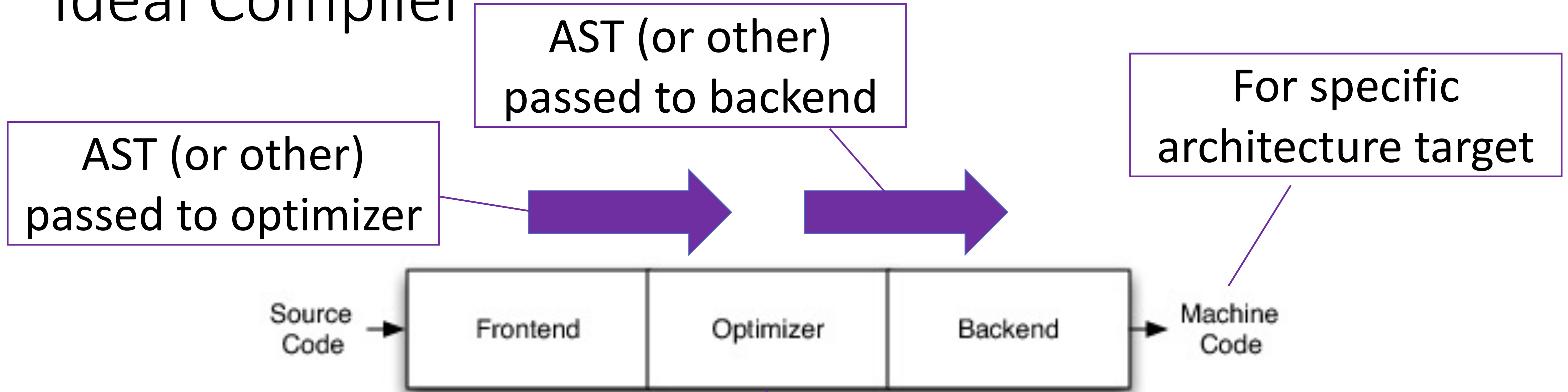
256-bit registers

```
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
```

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

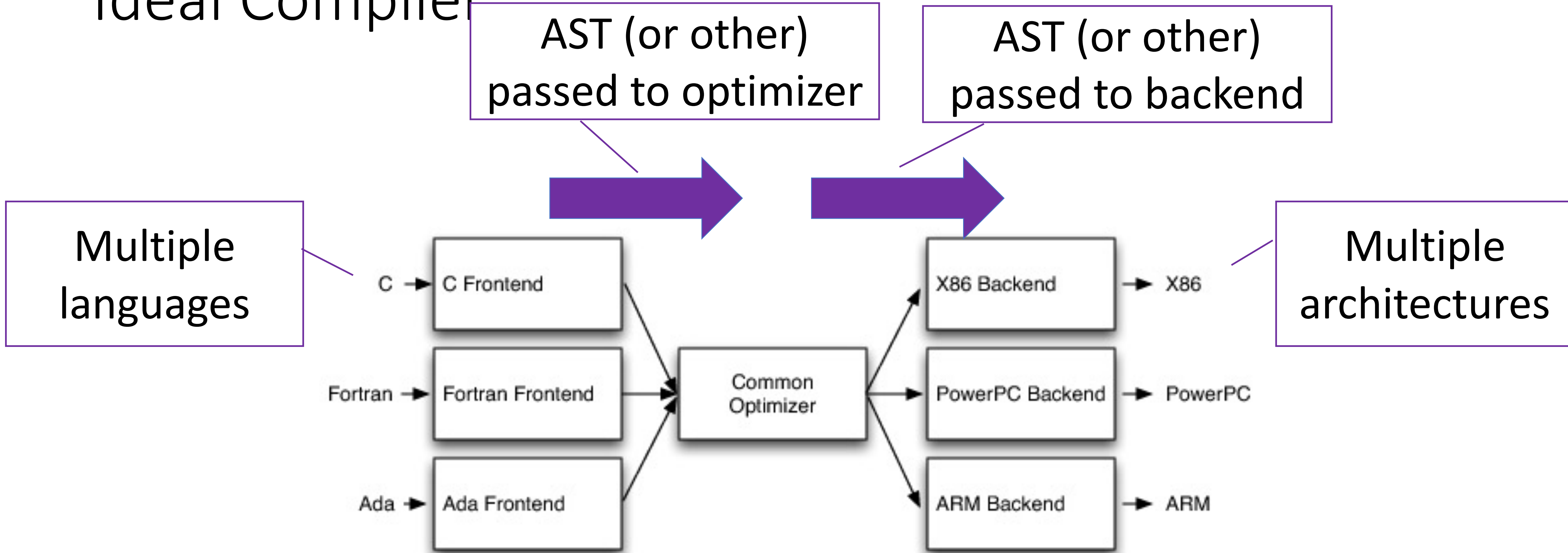


Generates abstract syntax tree (AST)

Language and architecture independent

How much effort for M languages and N architectures?

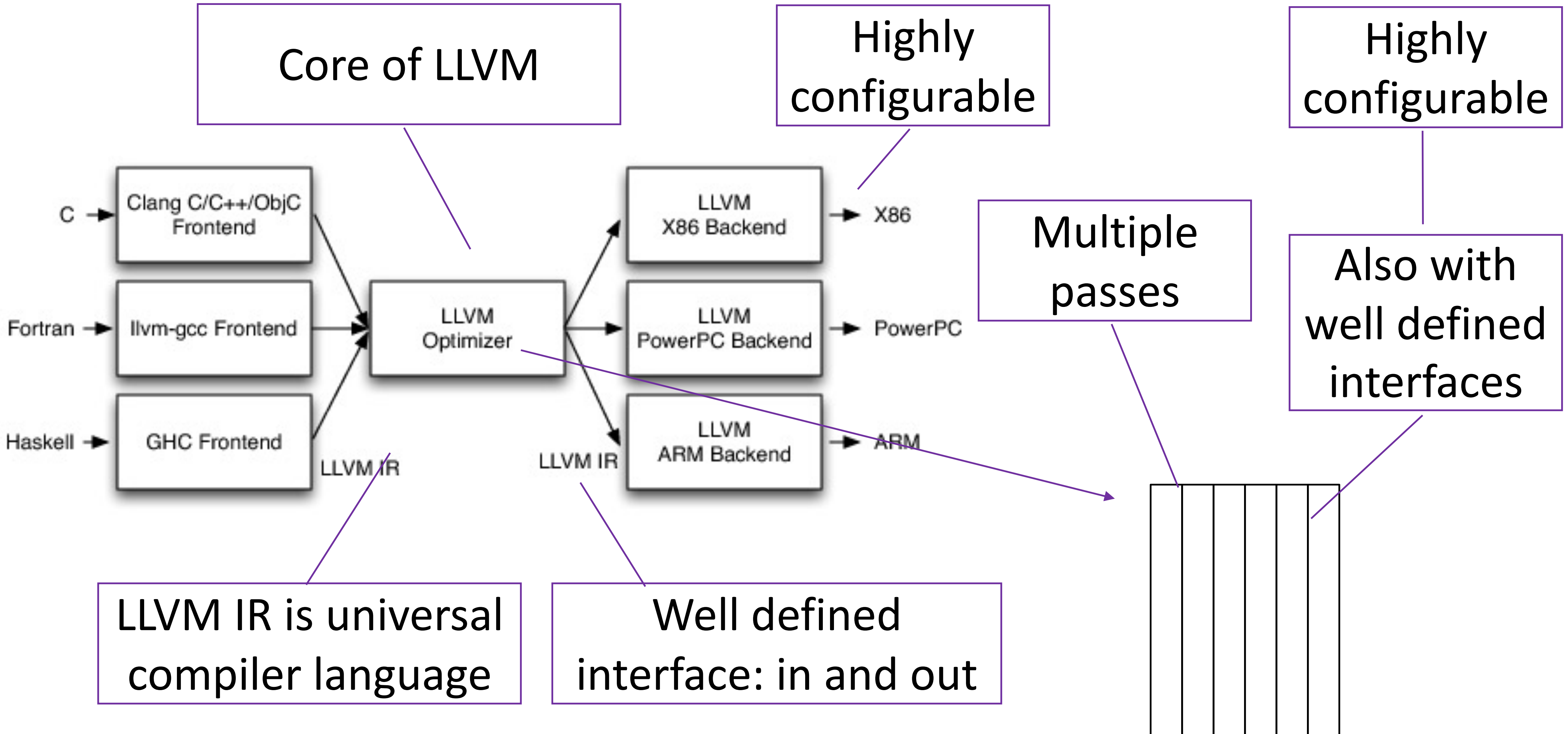
Ideal Compiler



How much effort for M languages and N architectures?

(In theory)

LLVM



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 -ma
vx2  -std=c++14 -Wc++14-extensions -fslp-vectorize-aggressive -mxsave -mavx -mavx2 -Wall - -o /d
ev/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" "-ma
in-file-name" "-" "-mrelocation-model" "pic" "-pic-level" "2" "-mthread-model" "posix" "-mdisabl
e-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" "-t
arget-feature" "-avx512ifma" "-target-feature" "-avx512dq" "-target-feature" "-fma4" "-target-fe
ature" "-prfchw" "-target-feature" "+bmi2" "-target-feature" "-xsavec" "-target-feature" "+fsgsb
ase" "-target-feature" "+popcnt" "-target-feature" "+aes" "-target-feature" "-pcommit" "-target-
feature" "-xsaves" "-target-feature" "-avx512er" "-target-feature" "-clwb" "-target-feature" "-a
vx512f" "-target-feature" "-pku" "-target-feature" "-smap" "-target-feature" "+mmx" "-target-fea
ture" "-xop" "-target-feature" "-rdseed" "-target-feature" "-hle" "-target-feature" "-sse4a" "-t
arget-feature" "-avx512bw" "-target-feature" "-clflushopt" "-target-feature" "-avx512vl" "-targe
t-feature" "+invpcid" "-target-feature" "-avx512cd" "-target-feature" "-rtm" "-target-feature" "
+fma" "-target-feature" "+bmi" "-target-feature" "-mwaitx" "-target-feature" "+rdrnd" "-target-f
eature" "+sse4.1" "-target-feature" "+sse4.2" "-target-feature" "+sse" "-target-feature" "+lzcnt
" "-target-feature" "+pclmul" "-target-feature" "-prefetchwt1" "-target-feature" "+f16c" "-targe
t-feature" "+ssse3" "-target-feature" "-sgx" "-target-feature" "+cmov" "-target-feature" "-avx51
2vbmi" "-target-feature" "+movbe" "-target-feature" "+xsaveopt" "-target-feature" "-sha" "-targe
t-feature" "-adx" "-target-feature" "-avx512pf" "-target-feature" "+sse3" "-target-feature" "+xs
ave" "-target-feature" "+avx" "-target-feature" "+avx2" "-target-linker-version" "278.4" "-dwarf
-column-info" "-debugger-tuning=lldb" "-resource-dir" "/Applications/Xcode.app/Contents/Develope
r/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/8.1.0" "-D" "NDEBUG" "-Ofast" "-Wc++1
4-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=m
acosx-10.12.0" "-fencode-extended-block-signature" "-fmax-type-align=16" "-fdiagnostics-show-opt
ion" "-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" "-d
emangle" "-lto_library" "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xcto
olchain/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" "/Ap
plications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang
/8.1.0/lib/darwin/libclang_rt.osx.a"
```

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) {
```

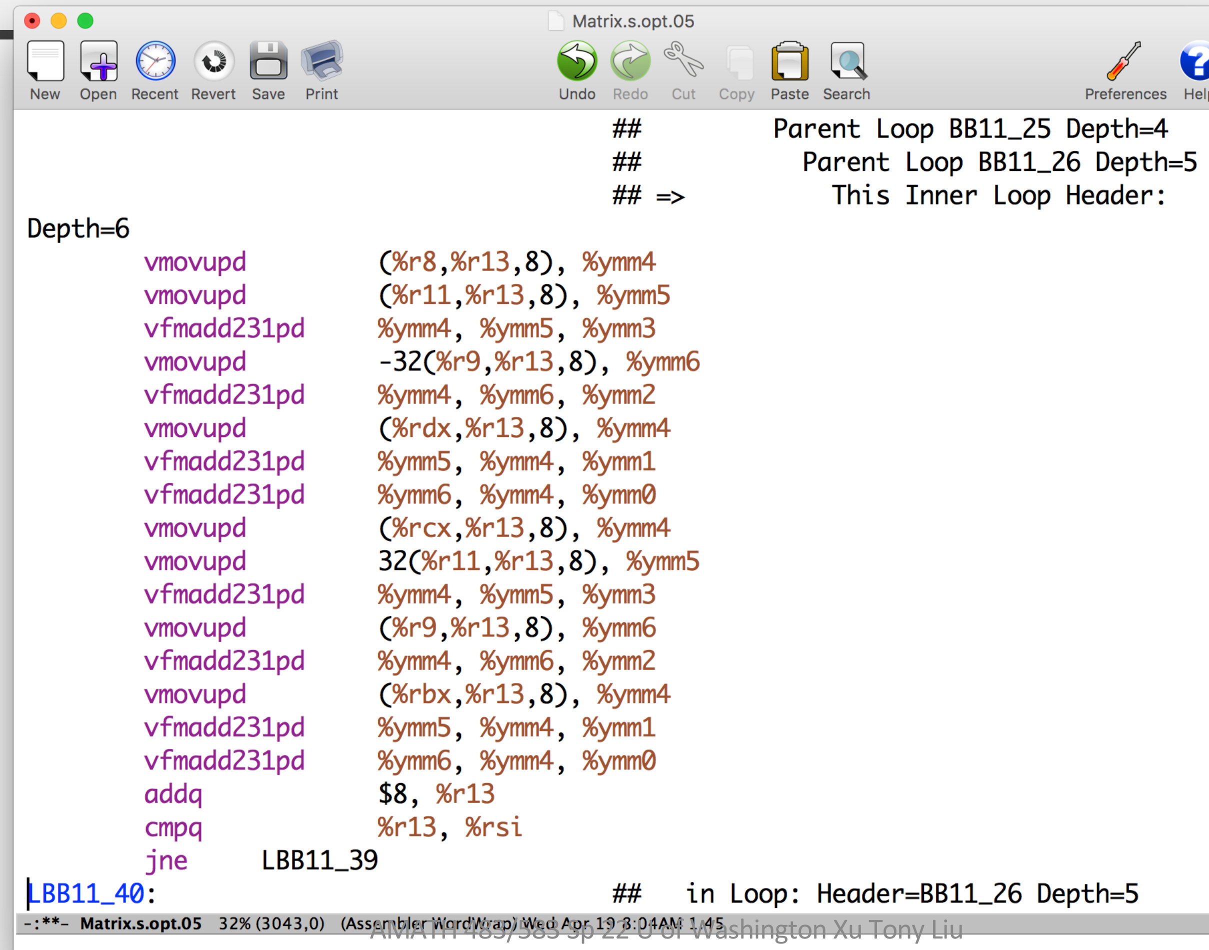
```
for (int j = 0; j < B.numCols(); ++j) {
  double t = C(i,j);
  for (int k = 0; k < A.numCols(); ++k) {
    t += A(i,k) * B(k,j);
  }
  C(i,j) = t;
}
```

Selects all

Unroll
Vectorization
Inline

As a Last Resort

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



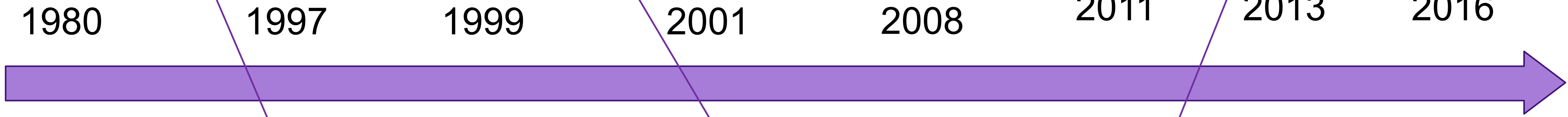
Advanced Vector Extensions

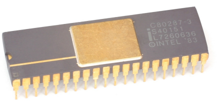
SIMD?

Multi Media Extensions

Streaming SIMD Extensions

Advanced Vector Extensions





8 80-bit stack registers

8 64-bit registers

8 128-bit registers (single) (xmm)

8 128-bit registers (double)

Many new instructions

16 256-bit registers (ymm)

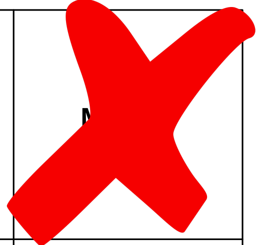
32 512-bit registers (zmm)

Flynn's Taxonomy (Aside)

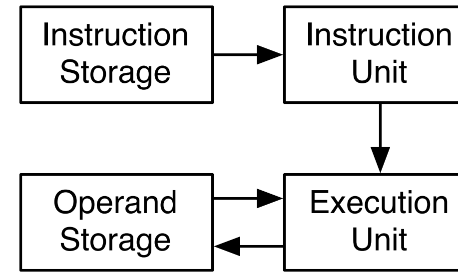
Anyone in HPC must know Flynn's taxonomy

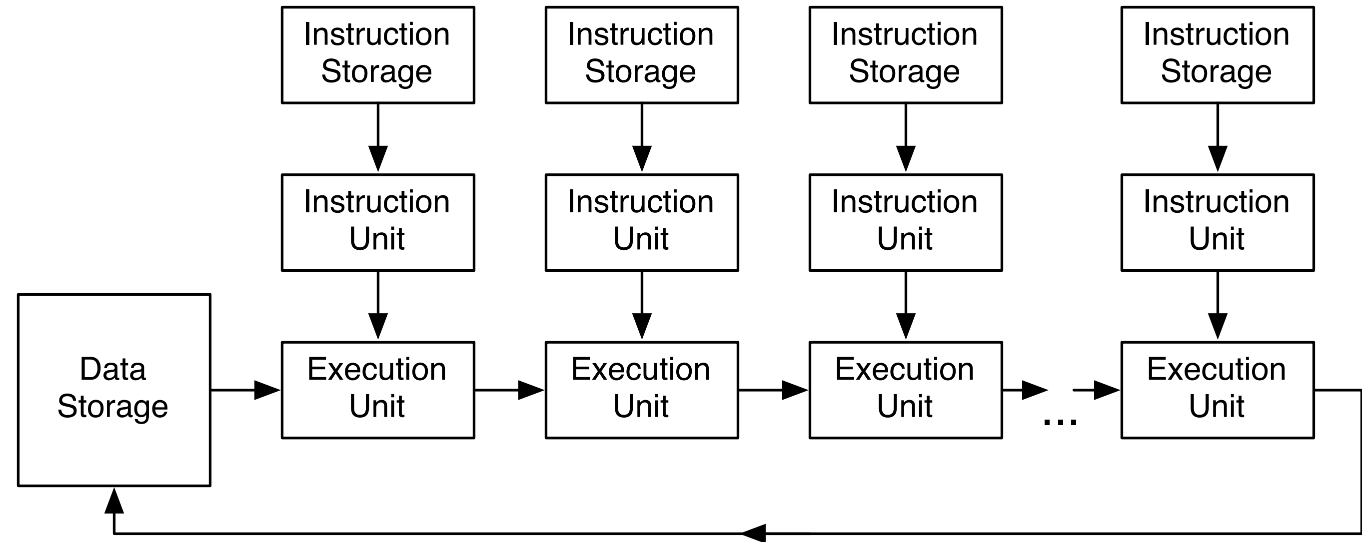
- **Classic** classification of parallel architectures (Michael Flynn, 1966)

Plain old sequential

	Single Instruction	Multiple Instruction
Single Data	SISD	
Multiple Data	SIMD	MIMD

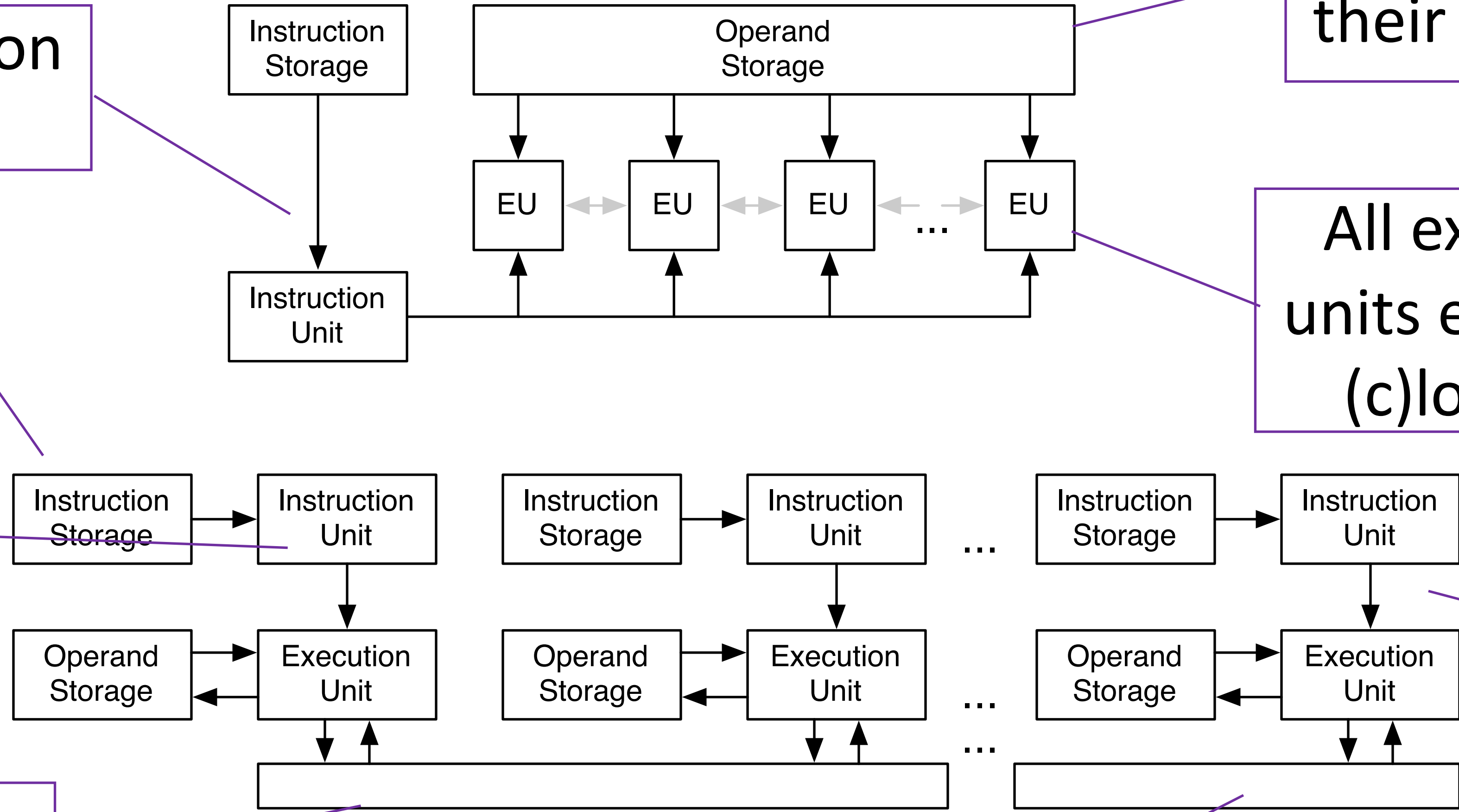
Based on multiplicity of instruction streams, data storage





SIMD and MIMD

- Two principal parallel computing paradigms (multiple op



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

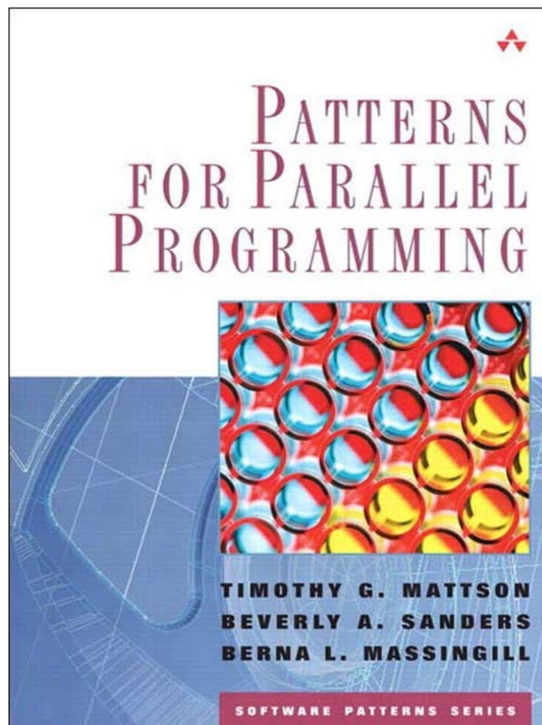
Shared Memory

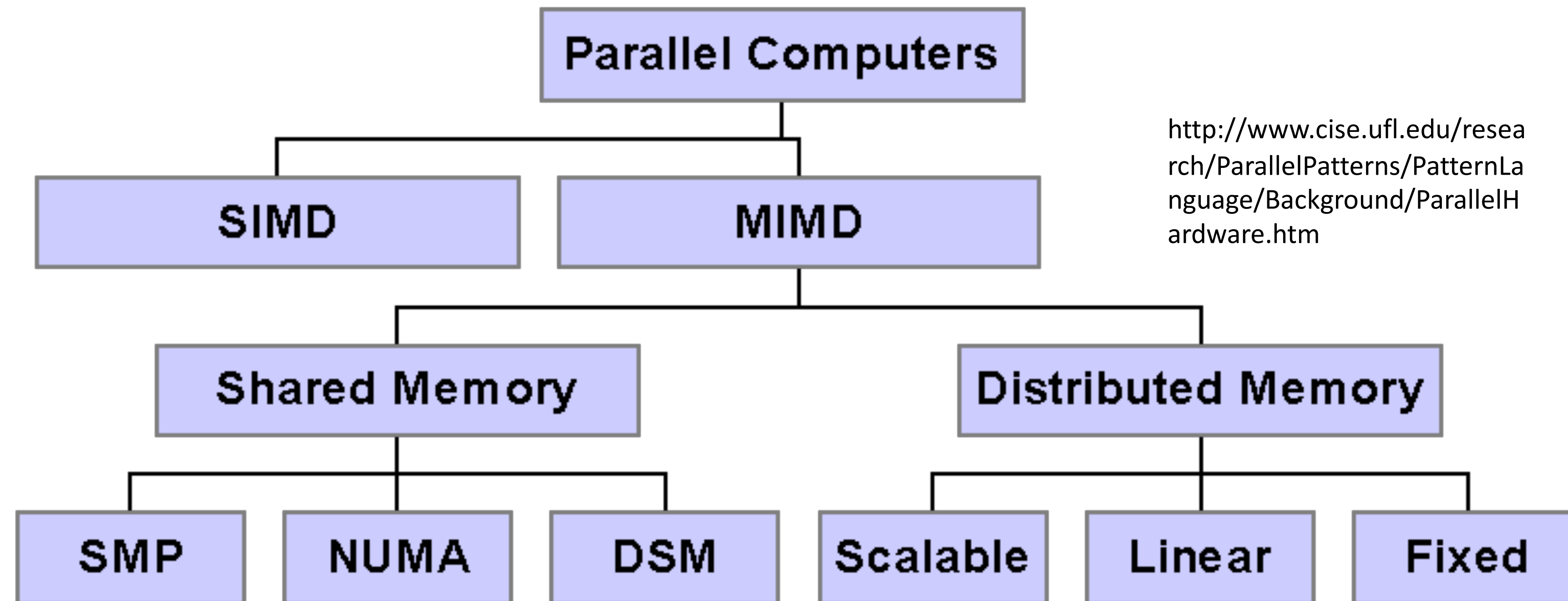
Not Shared

A More Refined (Programmer-Oriented) Taxonomy

- Three major modes … Memory
- Different programm… ed with different modes of parallelis… ted)
- A modern supercomputer will have all three major modes present

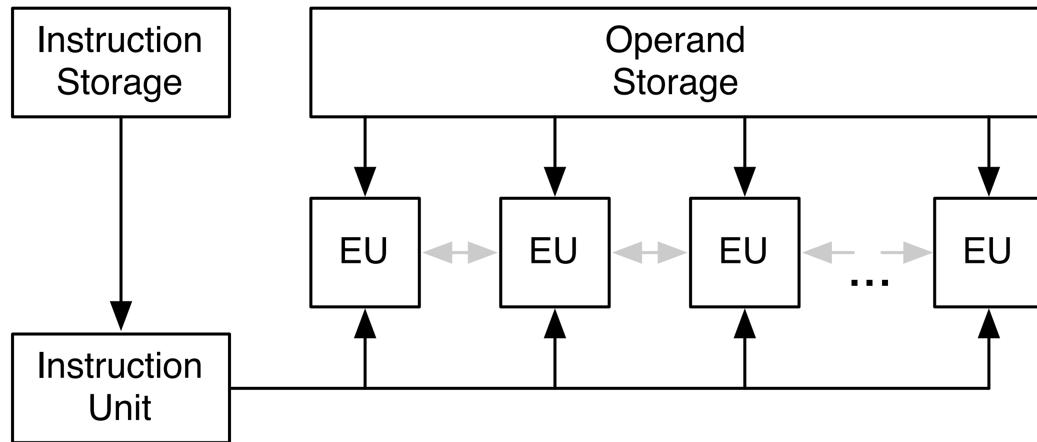
We will come back to this soon





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

SIMD in SSE/AVX

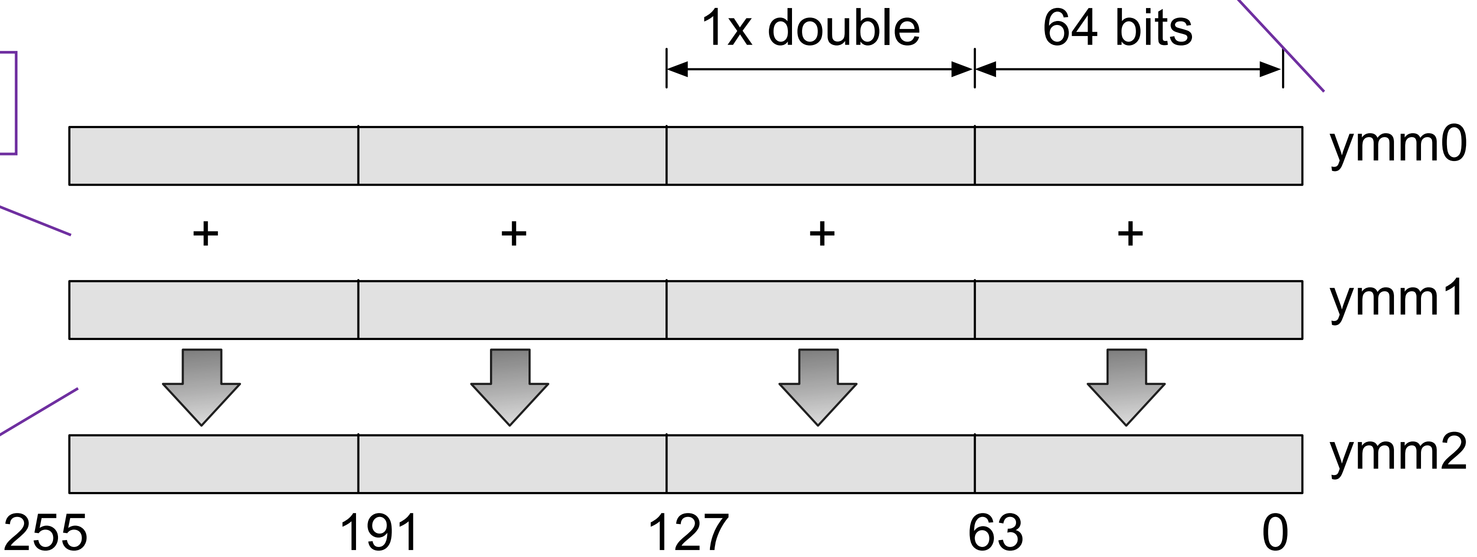


Flynn's original conceptual model

ymm are 256-bit registers

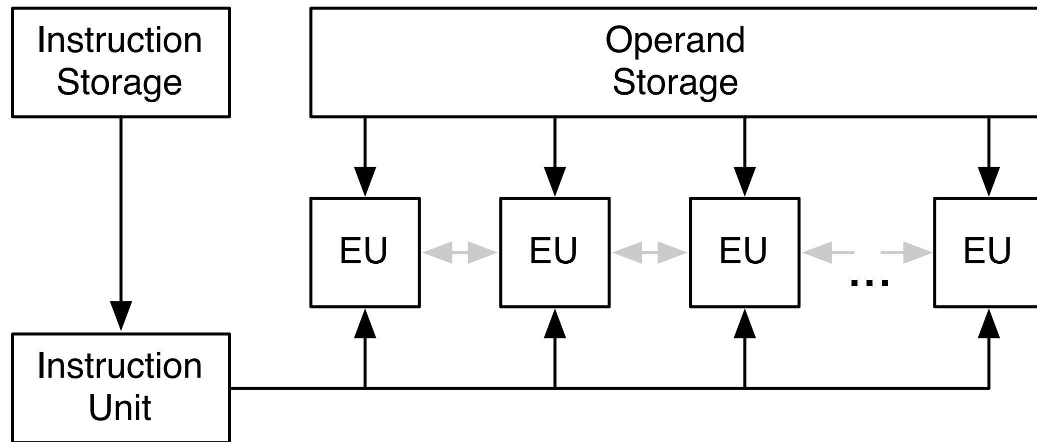
```
vfadd231pd  %ymm0, %ymm1, %ymm2
```

One machine instruction



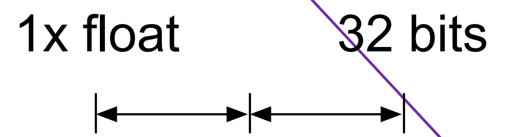
Adds all four doubles ***simultaneously***

SIMD in SSE/AVX

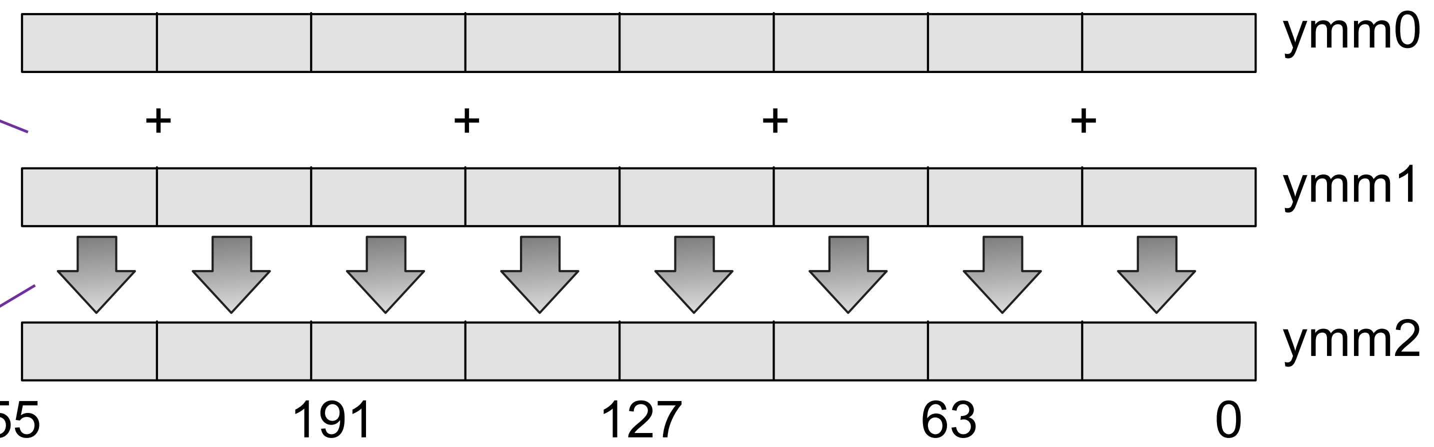


Flynn's original conceptual model

ymm are 256 bit registers



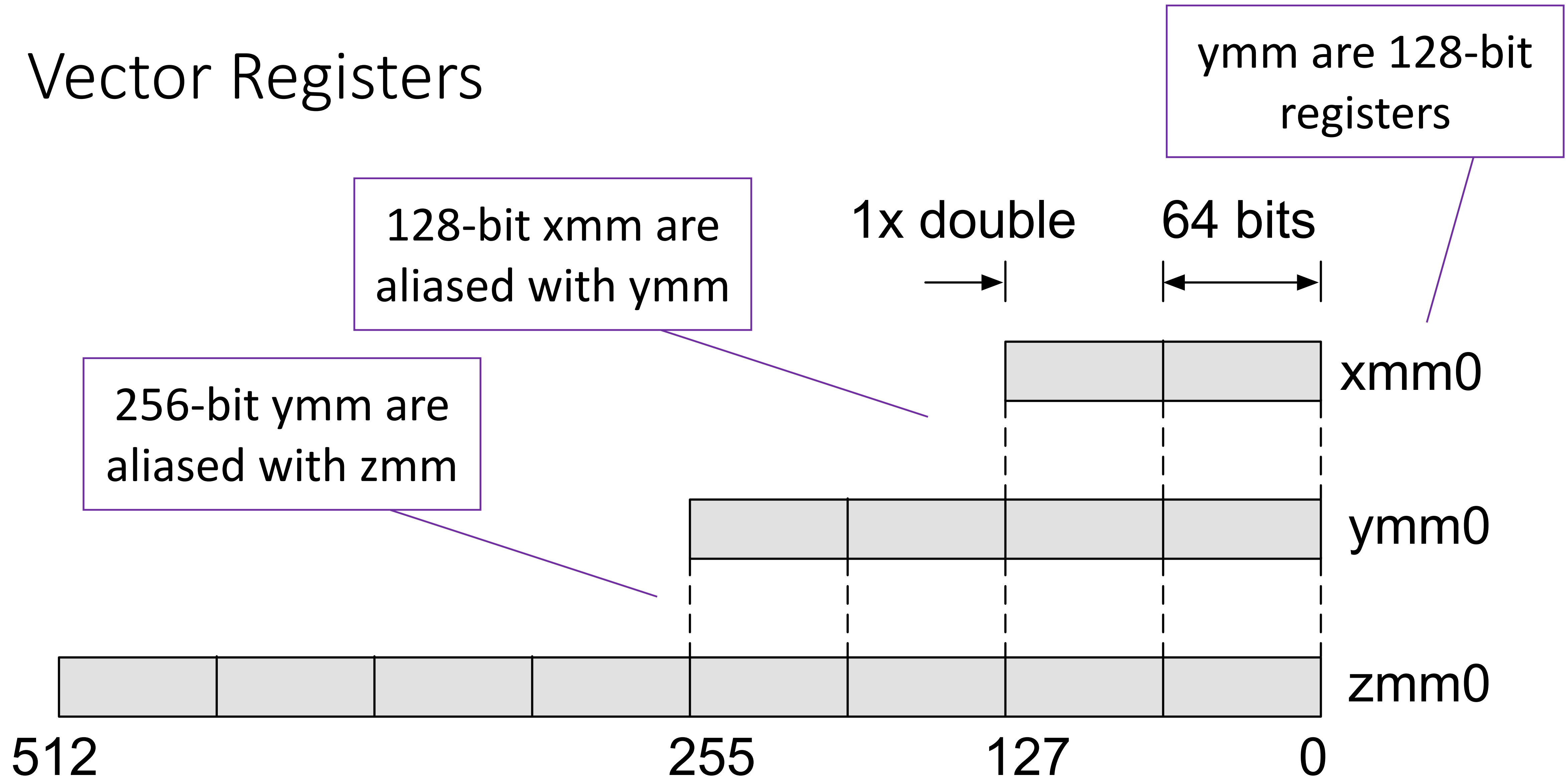

```
vfadd231ps  %ymm0, %ymm1, %ymm2
```



One machine instruction

Adds all eight floats ***simultaneously***

Vector Registers



Intel Intrinsics Guide

Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Technologies

- [x] MMX
- [ ] SSE
- [ ] SSE2
- [ ] SSE3
- [ ] SSSE3
- [ ] SSE4.1
- [ ] SSE4.2
- [ ] AVX
- [ ] AVX2
- [ ] FMA
- [ ] AVX-512
- [ ] KNC
- [ ] SVML
- [ ] Other

Categories

- [ ] Application-Targeted
- [x] Arithmetic
- [ ] Bit Manipulation
- [ ] Cast
- [ ] Compare

Intrinsic	Instruction
__m64 a, __m64 b)	paddw
__m64 _mm_add_pi32 (__m64 a, __m64 b)	paddd
__m64 _mm_add_pi8 (__m64 a, __m64 b)	paddb
__m64 _mm_adds_pi16 (__m64 a, __m64 b)	paddsw
__m64 _mm_adds_pi8 (__m64 a, __m64 b)	paddsb
__m64 _mm_adds_pu16 (__m64 a, __m64 b)	paddusw
__m64 _mm_adds_pu8 (__m64 a, __m64 b)	paddusb
__m64 _mm_madd_pi16 (__m64 a, __m64 b)	pmaddwd
__m64 _mm_mulhi_pi16 (__m64 a, __m64 b)	pmulhw
a, __m64 b)	pmullw
m64 b)	paddb
m64 b)	paddd
m64 b)	paddsb
__m64 _m_paddsw (__m64 a, __m64 b)	paddsw
__m64 _m_paddusb (__m64 a, __m64 b)	paddusb
__m64 _m_paddusw (__m64 a, __m64 b)	paddusw

Choose family

Get back intrinsics

Choose operation

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

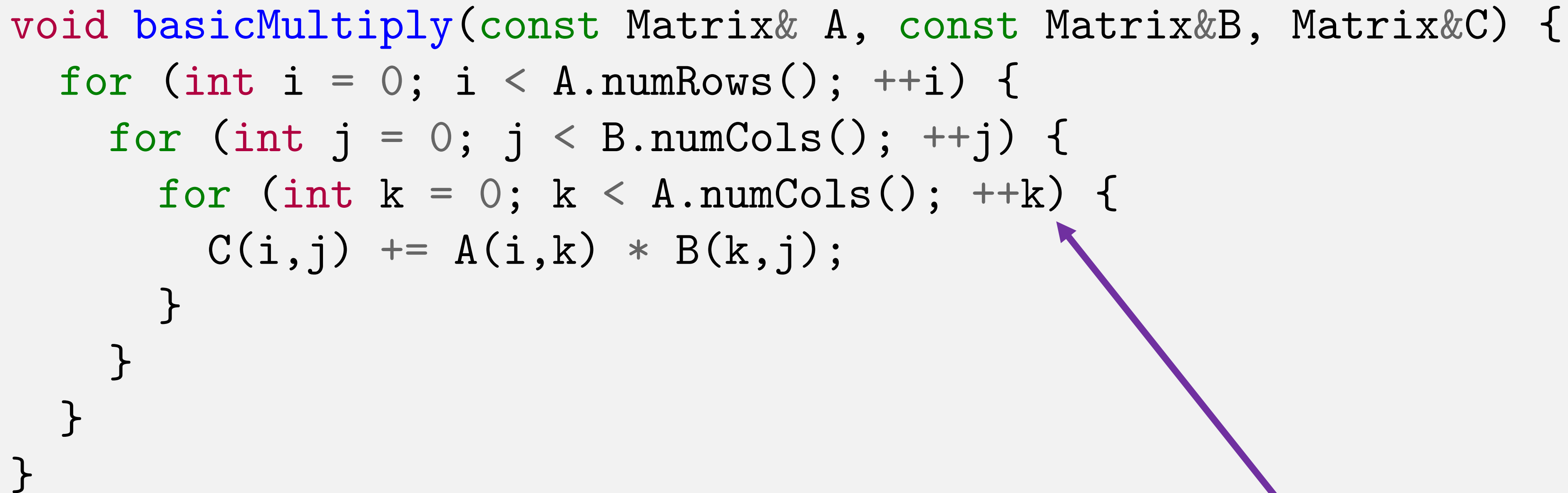
cpuid instruction

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?




```
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

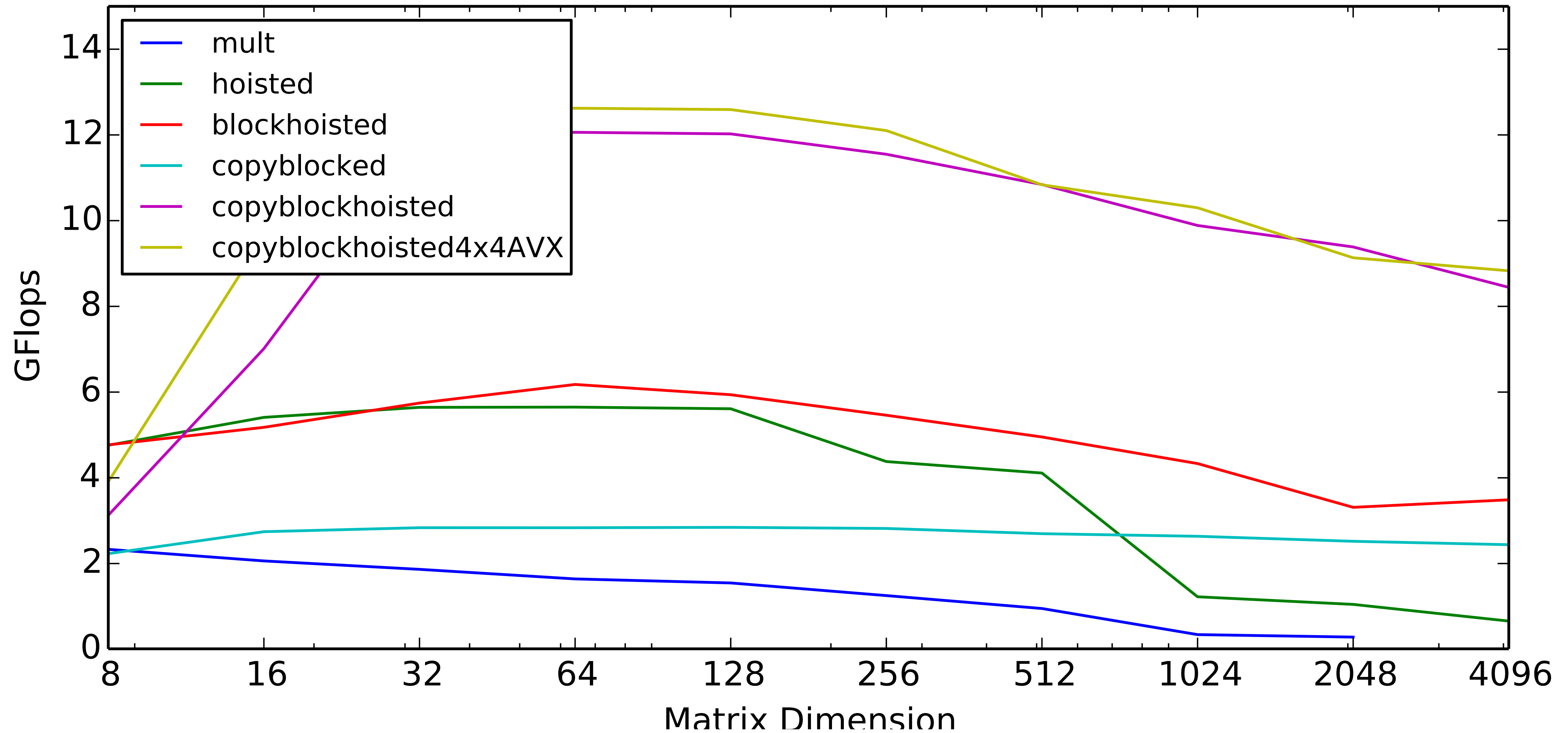
```
movq     -8(%rbp), %rdi
movslq   -28(%rbp), %rsi
movslq   -36(%rbp), %rdx
callq    __ZNK6MatrixclEmm
movsd    (%rax), %xmm0
movq     -16(%rbp), %rdi
movslq   -36(%rbp), %rsi
movslq   -32(%rbp), %rdx
movsd    %xmm0, -72(%rbp)
callq    __ZNK6MatrixclEmm
movsd    -72(%rbp), %xmm0
mulsd    (%rax), %xmm0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



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

Thank You!

Creative Commons BY-NC-SA 4.0 License

© Andrew Lumsdaine, 2017-2022

Except where otherwise noted, this work is licensed under

https://creativecommons.org/licenses/by-nc-sa/4.0/