AMATH 483/583
High Performance Scientific Computing

Lecture 8:
BLAS, Strassen's Algorithm, Roofline Model

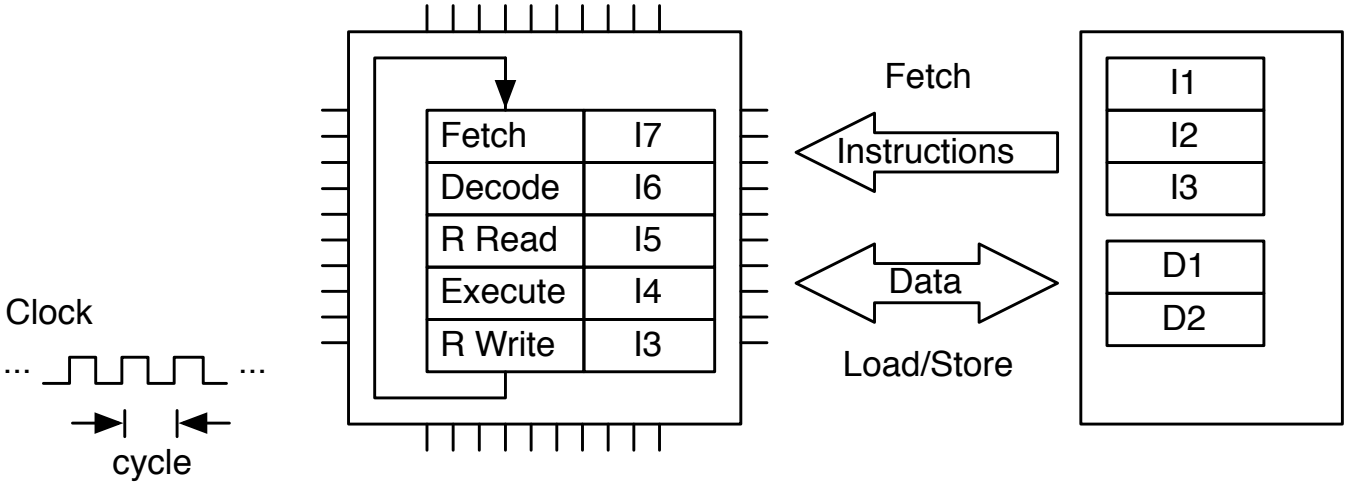
Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

- Finish SIMD
- Revisit pipelining, hoisting, copying
- BLAS
- Strassen's Algorithm
- Roofline Model

Processor Core Instruction Handling

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



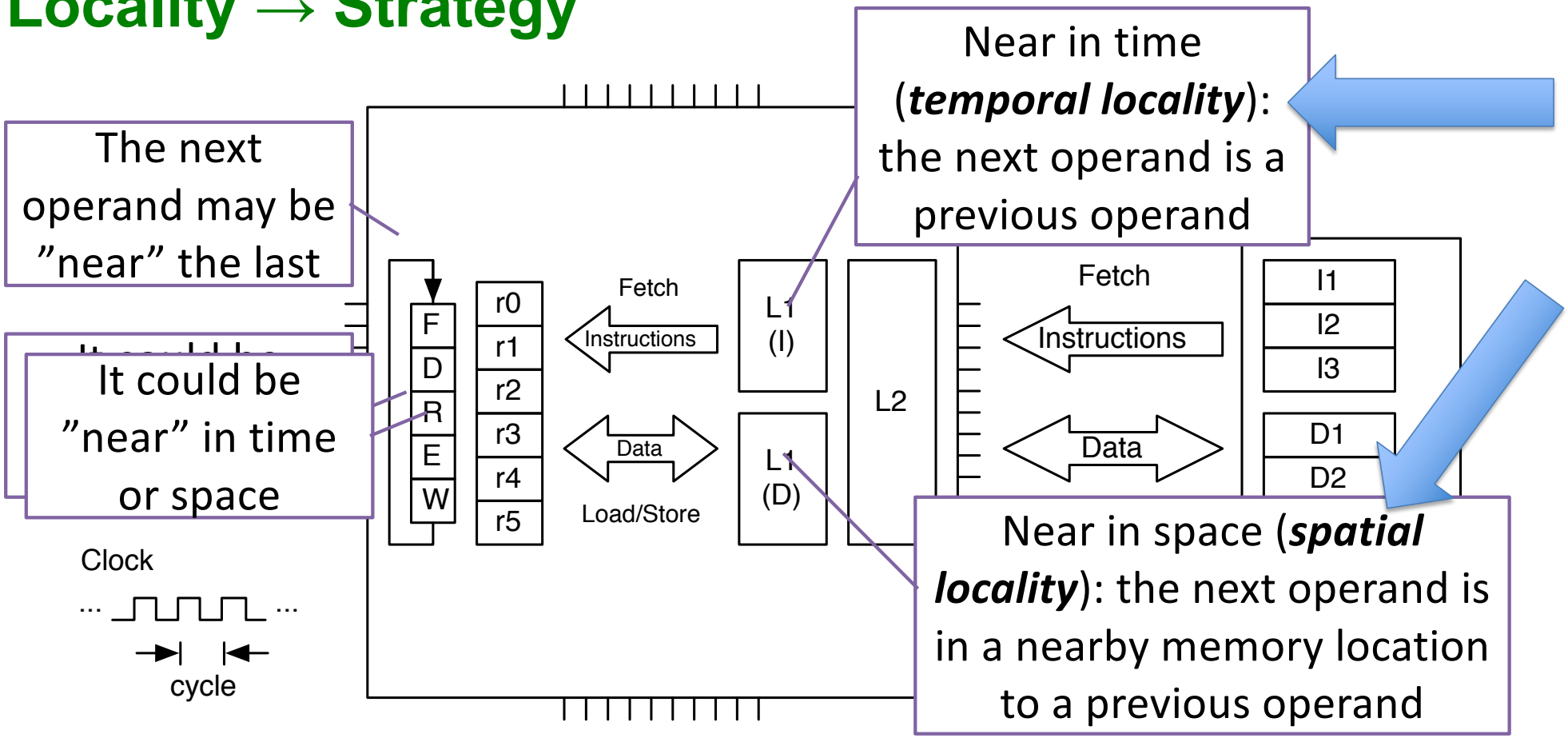
Performance-Oriented Architecture Features

- 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

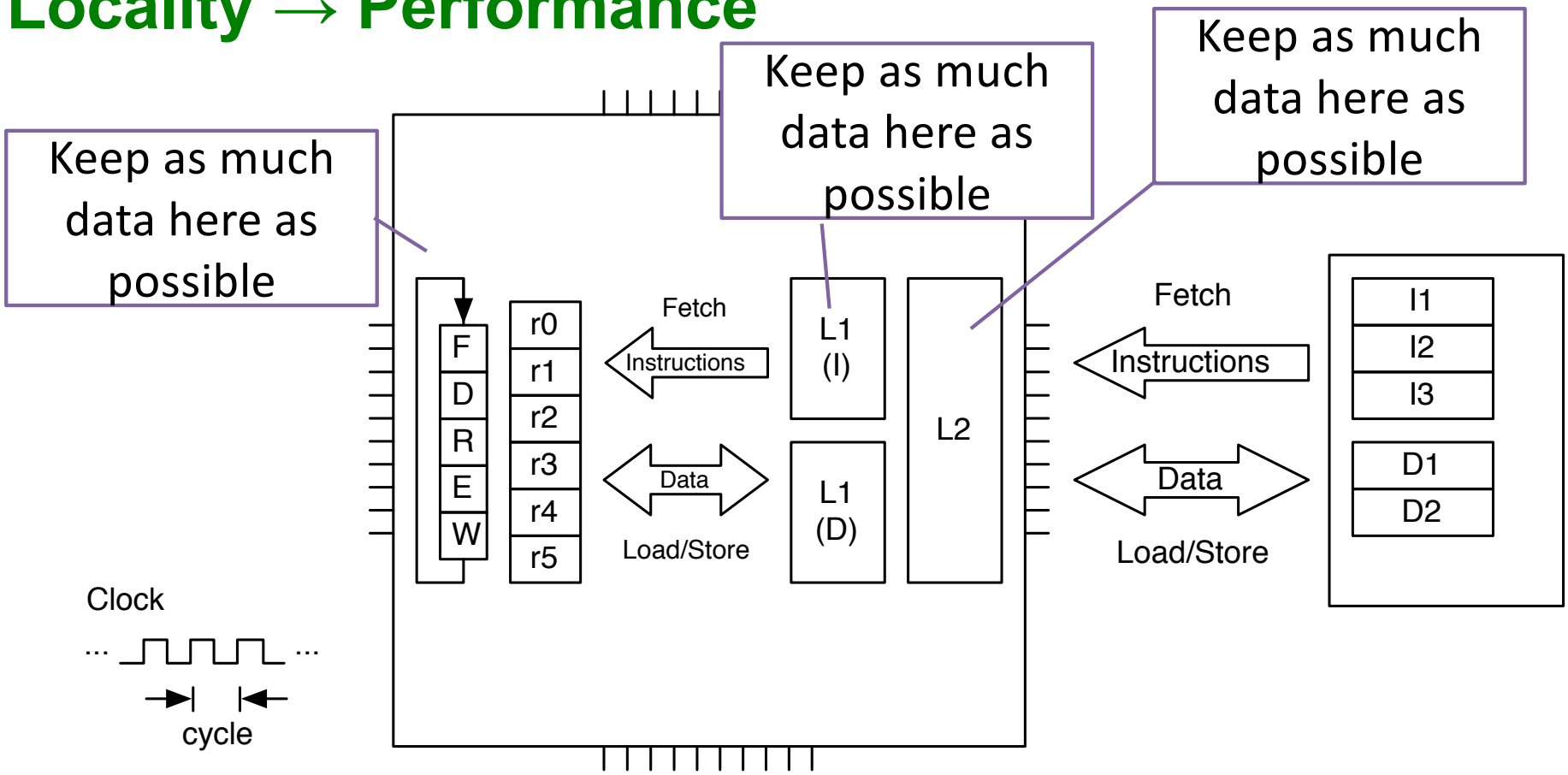
Unrolling and inlining can help these

But compilers are really good at that

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()

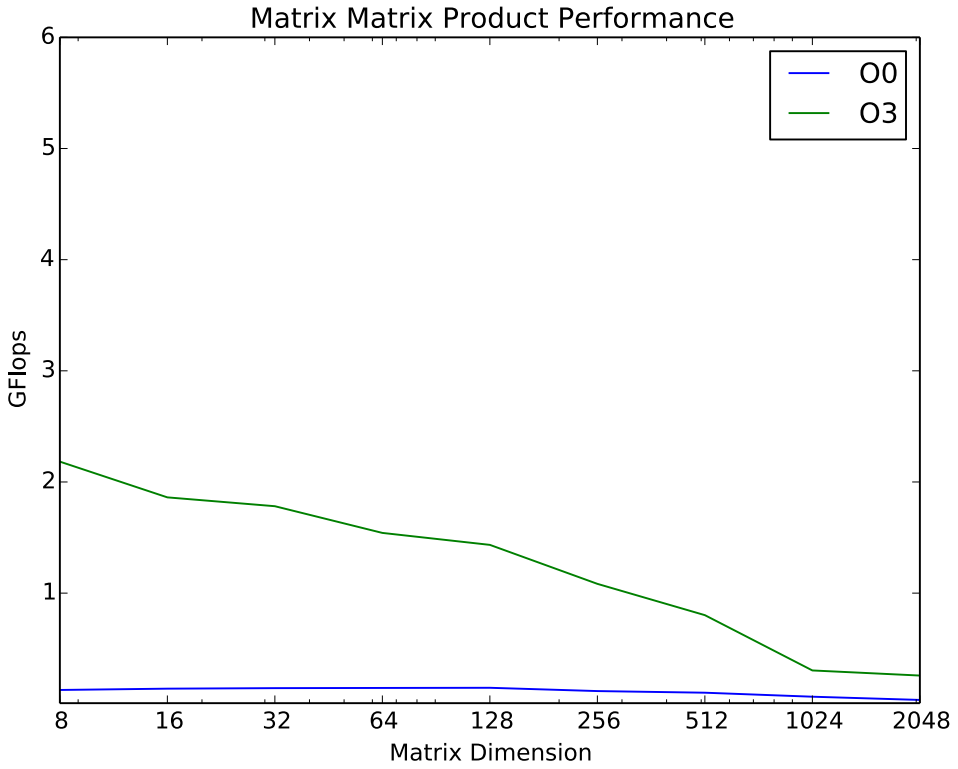
Just For Benchmarking

```
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

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)

Why not automatically?

```
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)

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

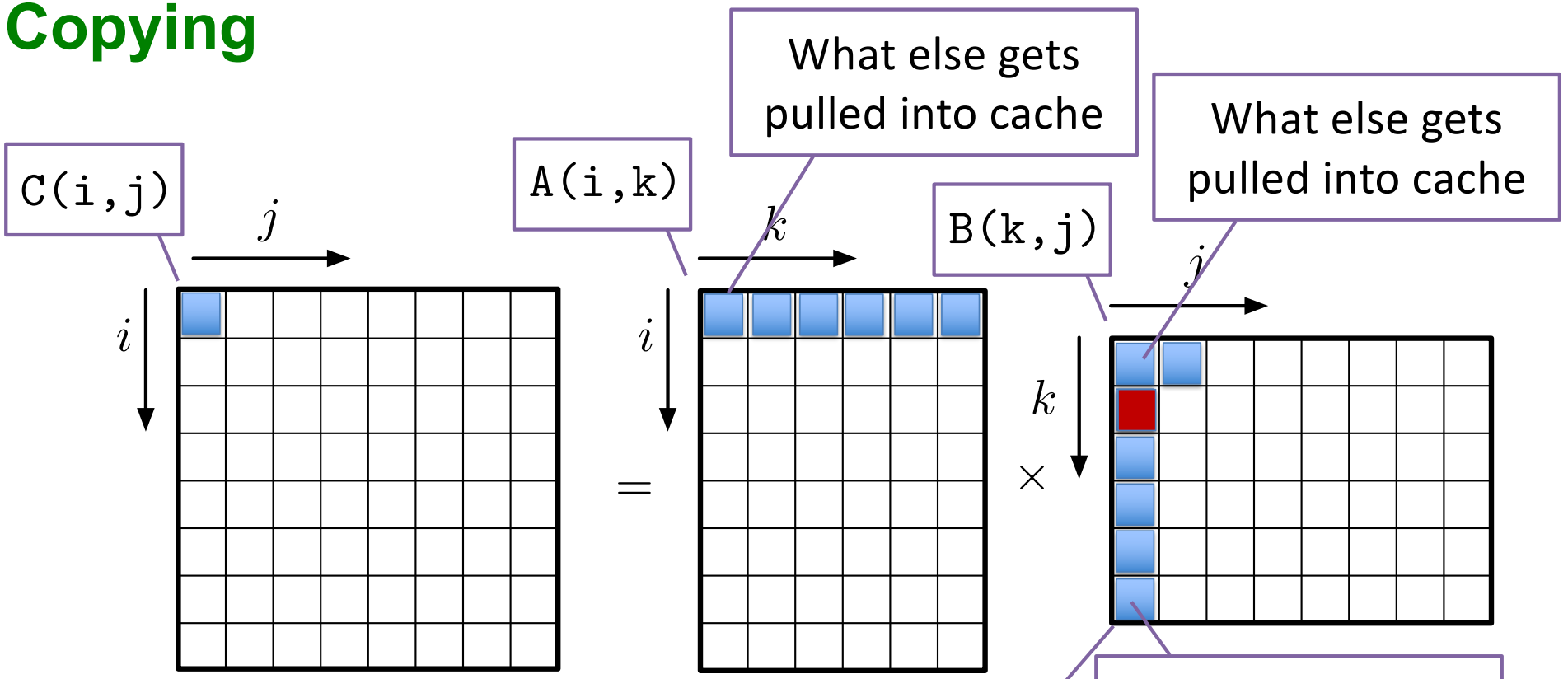
A(i,k) is used twice

Can also hoist (independent of k)

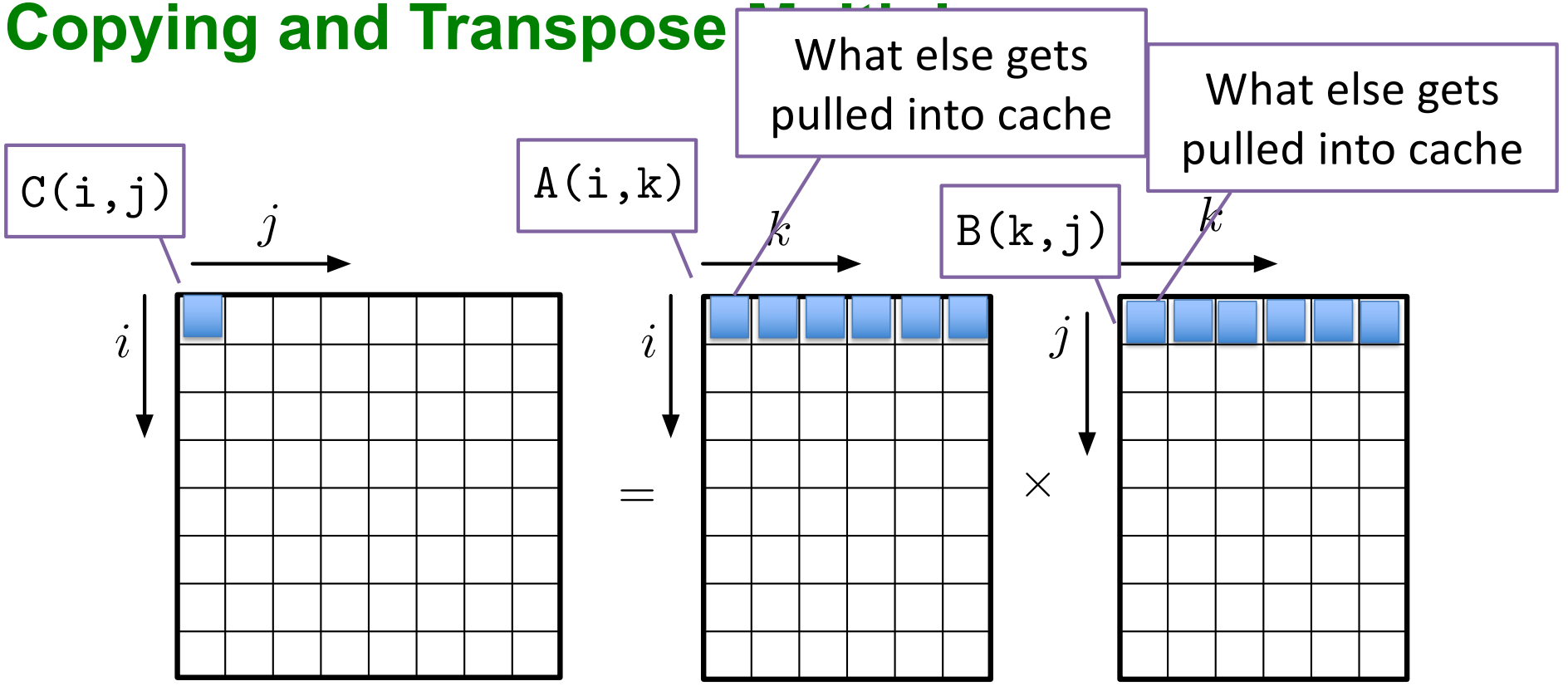
A(i+1,k) is used twice

- 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

Copying


C(i,j)
A(i,k)
B(k,j)
What else gets pulled into cache
What else gets pulled into cache
Or row order of transpose of B
Column order B
Want indexing in order of majority

Copying and Transpose

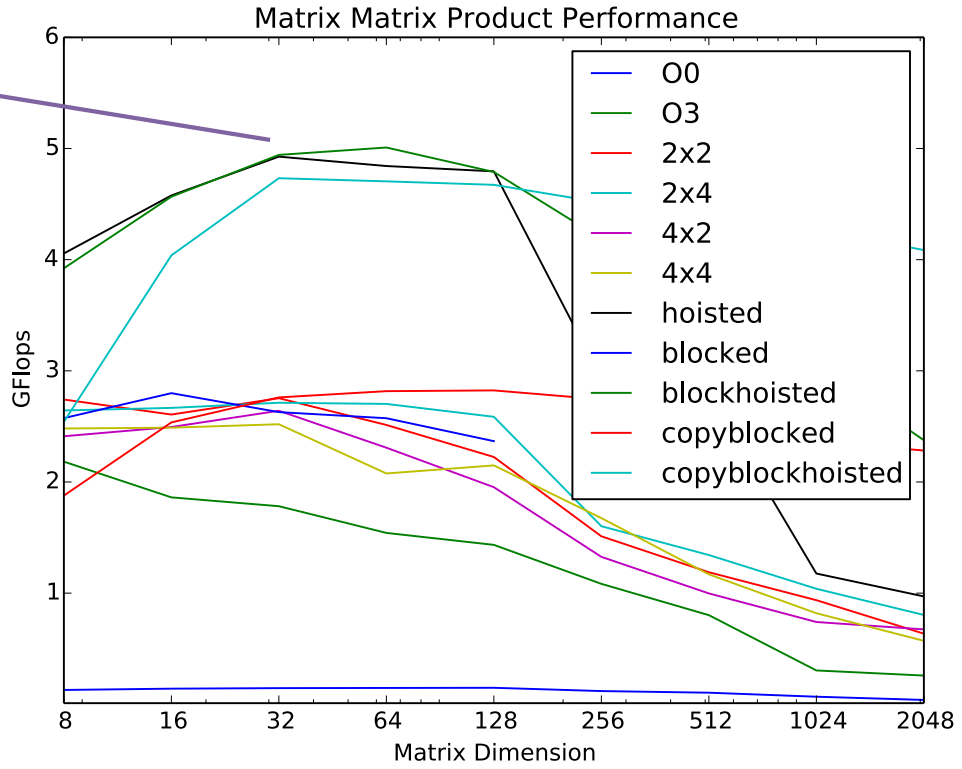


Blocking and Tiling and Hoisting and Copying

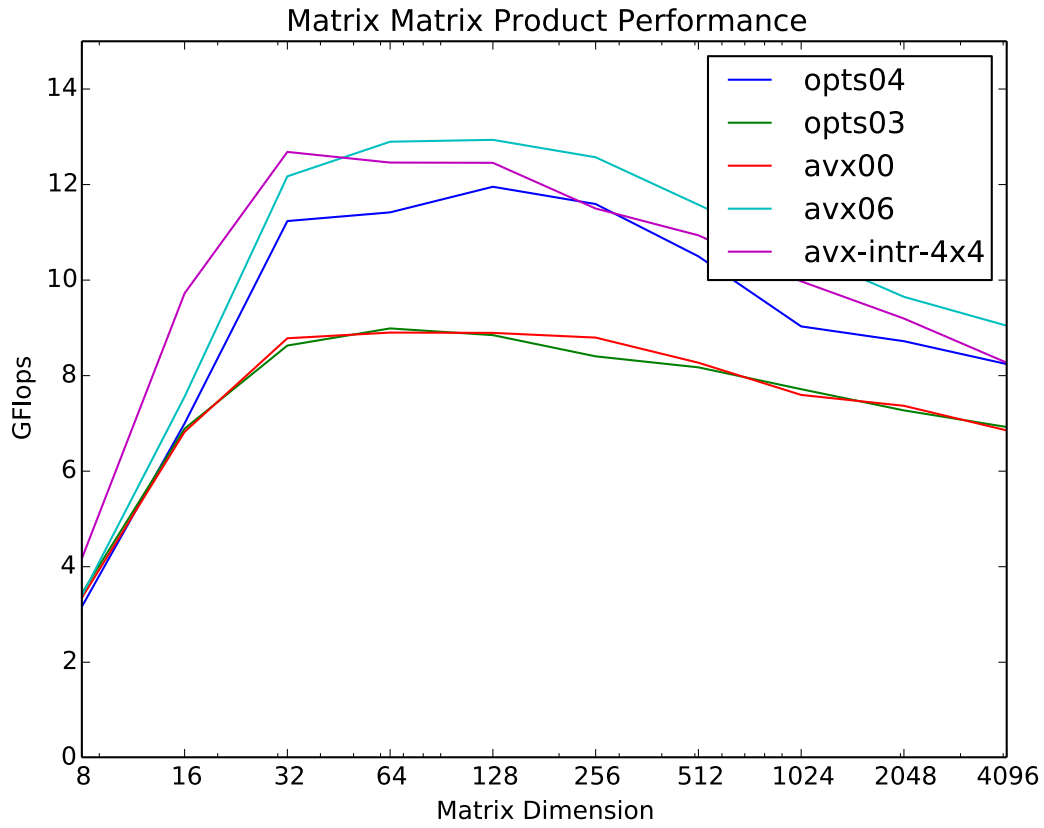
Is this the best we can do?

How good is it anyway?

(cf PS 4A)



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);
  }
}
```

X86 Assembly

AVX instructions

256 bit register

```
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
```

Fused Multiply-Add

Multiply-Add are separate here

8 FLOPS per cycle?

Vector Operations from C++

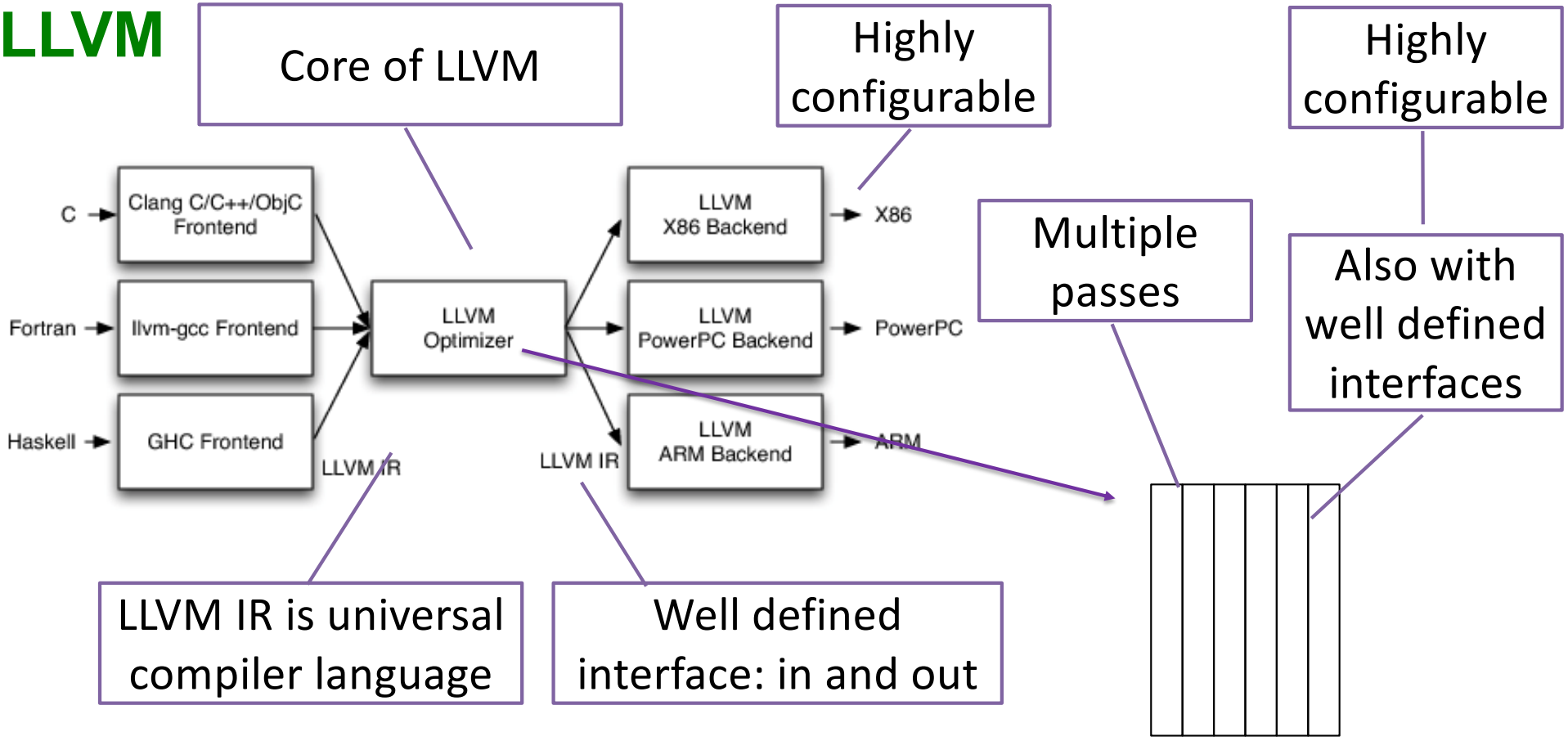
```
for (int i = ii; i < ii+blocksize; i += 2) {
  for (int j = jj, jb = 0; j < jj+blocksize; j += 2, jb += 2) {
    double t00 = C(i,j);        double t01 = C(i,j+1);
    double t10 = C(i+1,j);      double t11 = C(i+1,j+1);

    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
```

LLVM


Core of LLVM
Highly configurable
Highly configurable
Multiple passes
Also with well defined interfaces
C
Clang C/C++/ObjC Frontend
Fortran
llvm-gcc Frontend
Haskell
GHC Frontend
LLVM Optimizer
LLVM X86 Backend
LLVM PowerPC Backend
LLVM ARM Backend
X86
PowerPC
ARM
LLVM IR
LLVM IR
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 -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) {
```

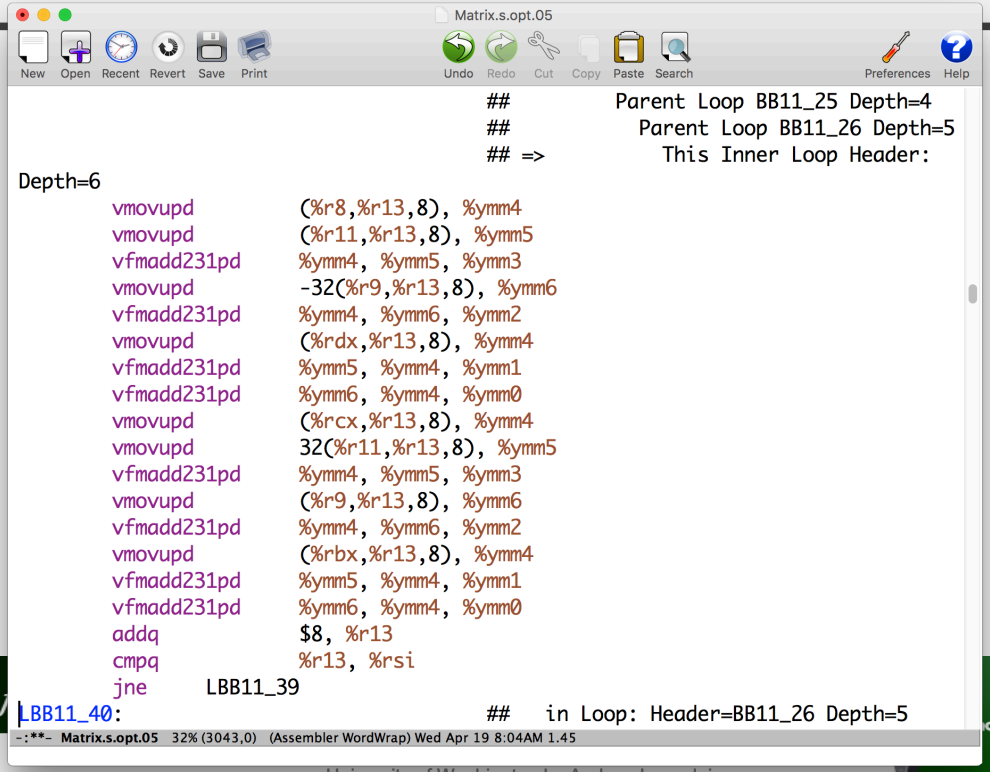
```
      for (int j = 0; j < B.numCols(); ++j) {
        double t = C(i,j);
        for (int k = 0; k < A.numCols(); ++k) {
          t += A(i,k) * B(k,j);
        }
        C(i,j) = t;
      }
```

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

1980 1997 1999 2001 2004 2006 2007 2008 2011 2013 2016

8087 MMX SSE SSE2 SSE3 SSE3.1 SSE4.1 SSE4.2 AVX AVX2 AVX512

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)

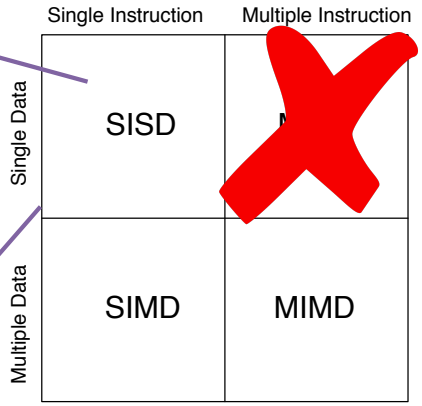
32 512-bit registers (zmm)

Flynn's Taxonomy (Aside)

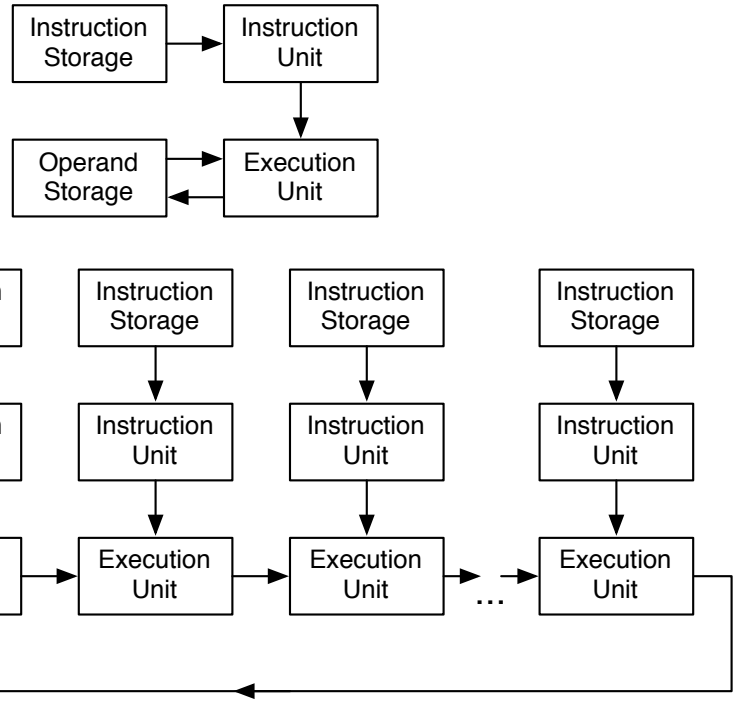
Anyone in HPC must know Flynn's taxonomy

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

Plain old sequential



Based on multiplicity of instruction streams, data storage

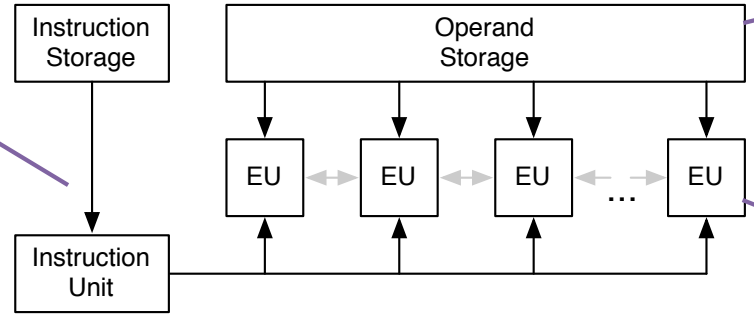


SIMD and MIMD

- Two principal parallel computing paradigms (multiple c

Single instruction at a time

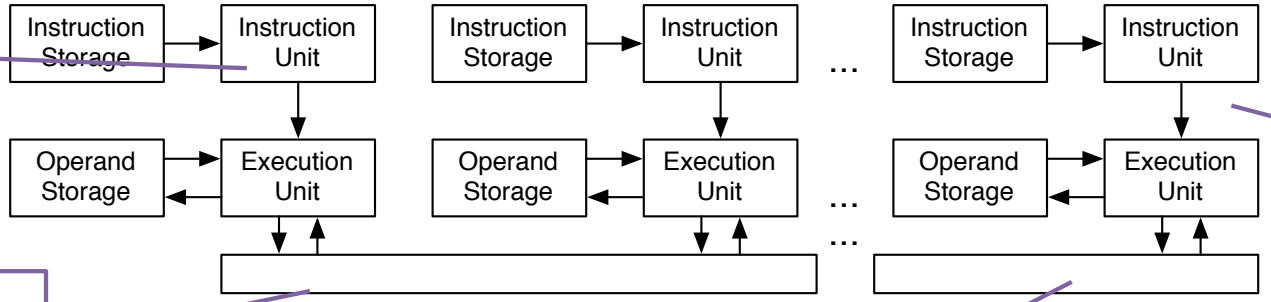
But each have their own data



Multiple instruction

All execution units execute in (c)lock step

EUs run independently (w own instrs)



Coming up next

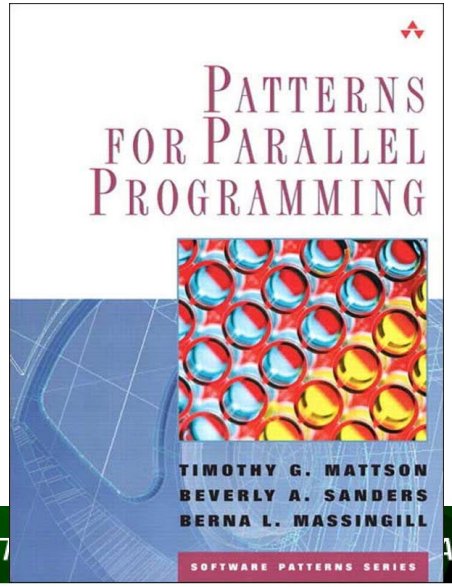
Shared Memory

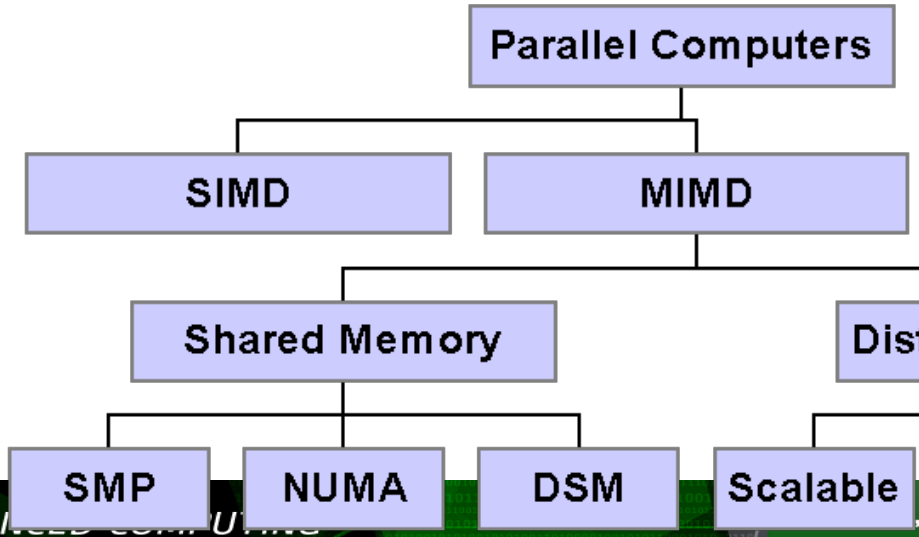
Not Shared

A More Refined (Programmer-Oriented) Taxonomy

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

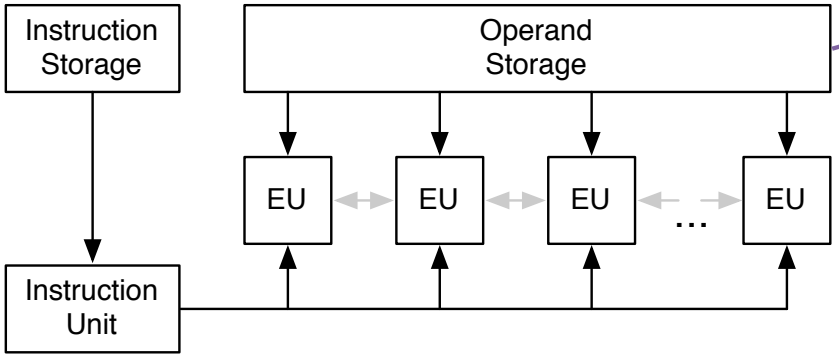
We will come back to this soon





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

SIMD in SSE/AVX

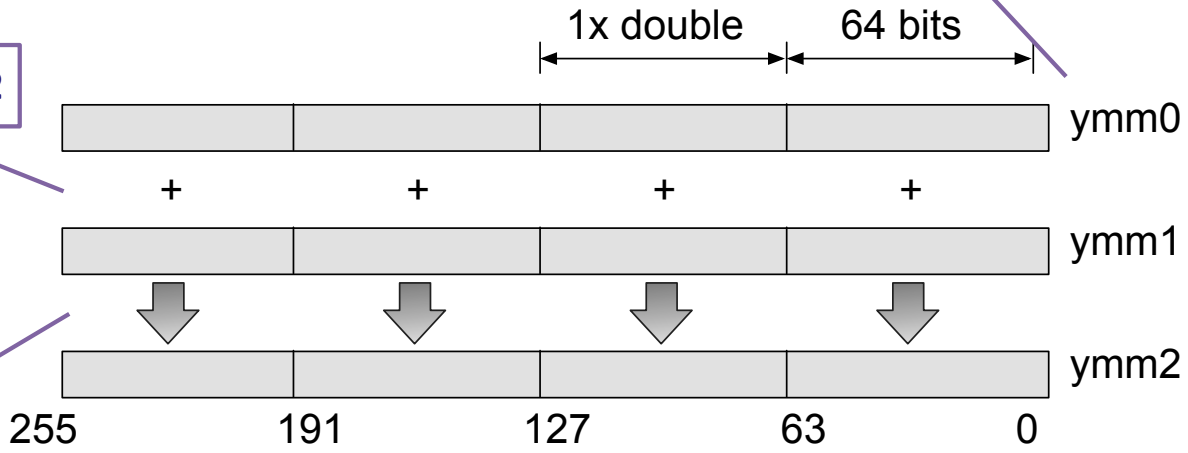


Flynn's original conceptual model

ymm are 256 bit registers

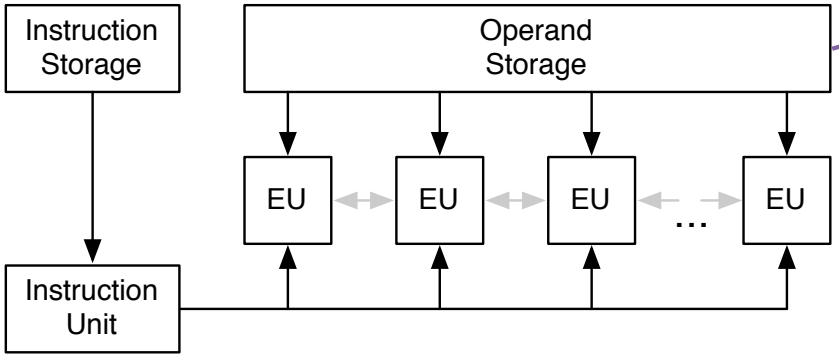
```
vfadd231pd  %ymm0, %ymm1, %ymm2
```

One machine instruction



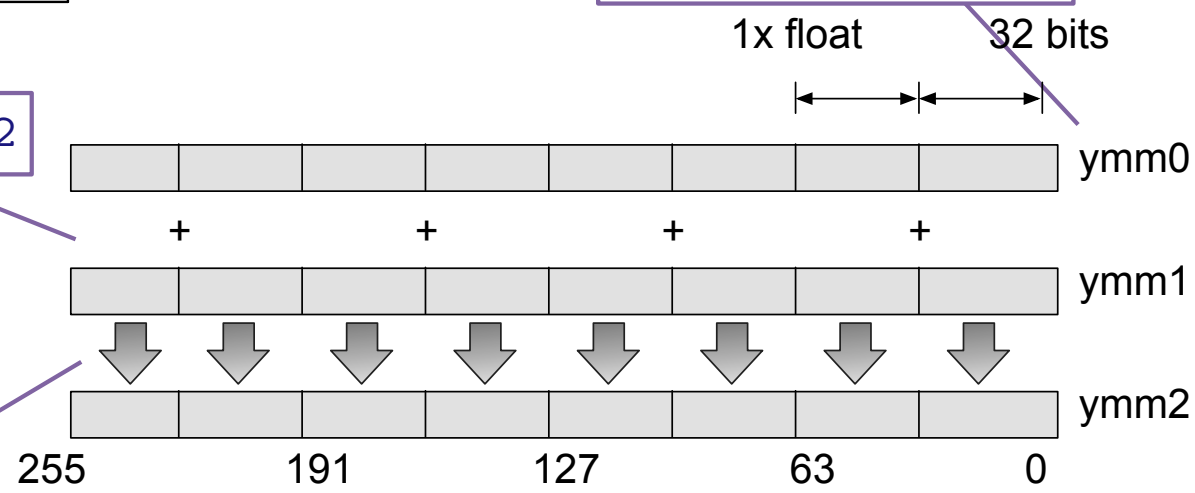
Adds all four doubles ***simultaneously***

SIMD in SSE/AVX



Flynn's original conceptual model

ymm are 256 bit registers

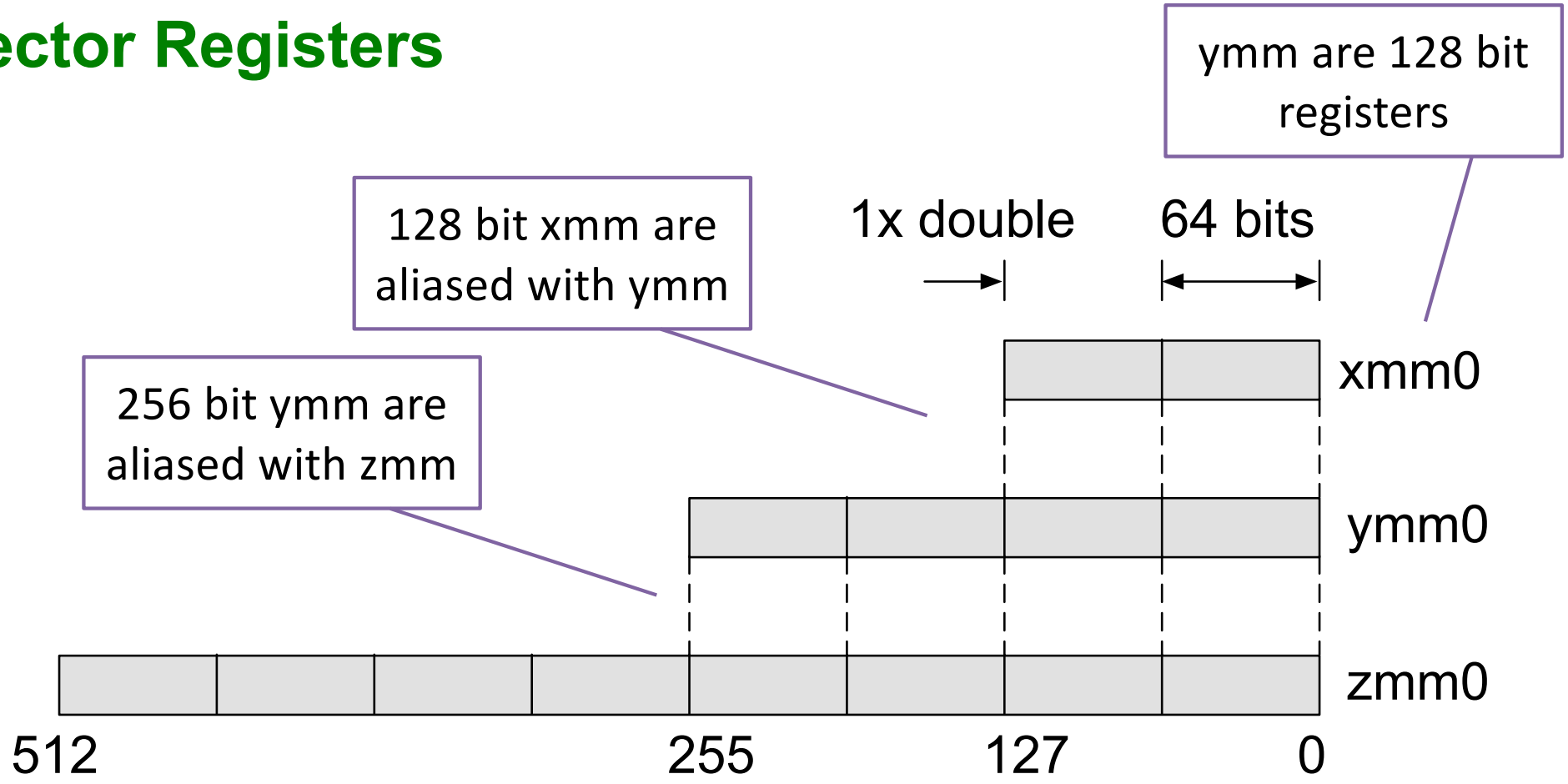



```
vfadd231ps  %ymm0, %ymm1, %ymm2
```

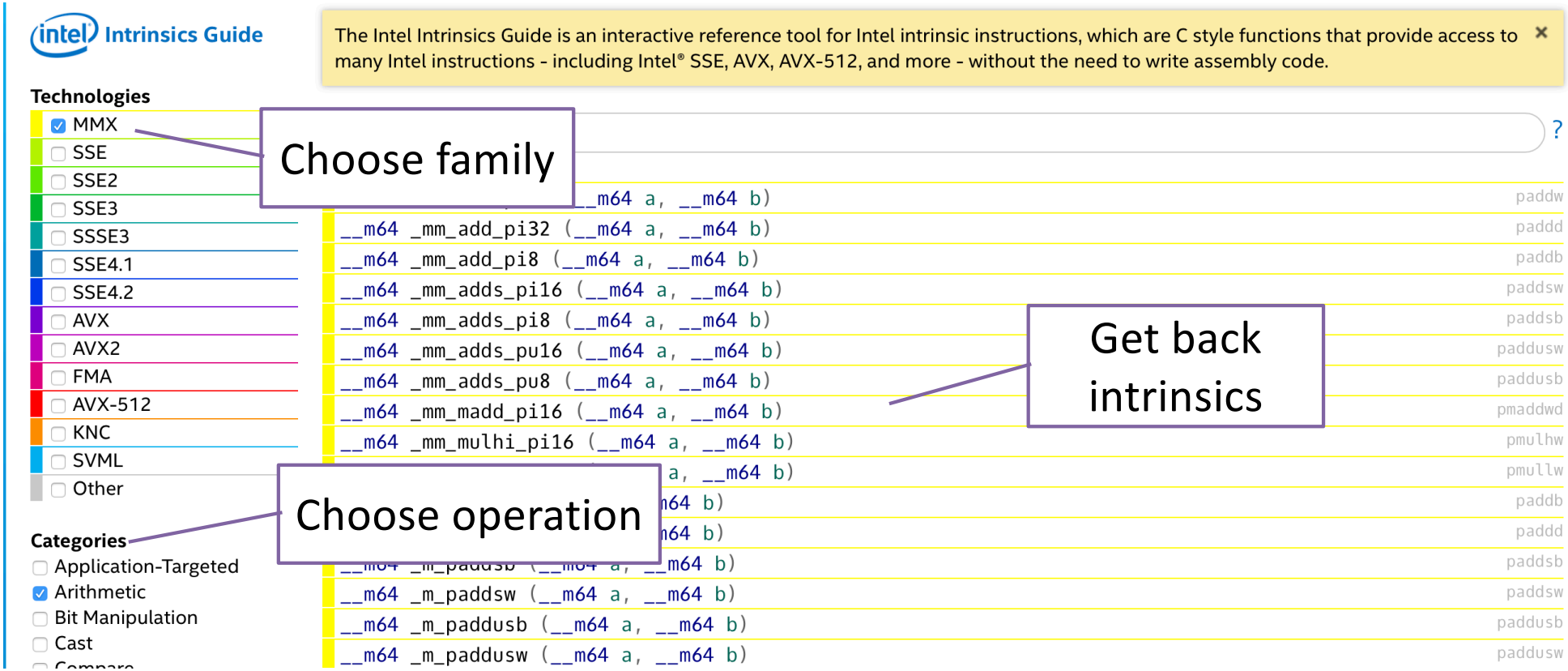
One machine instruction

Adds all eight floats ***simultaneously***

Vector Registers



Intel Intrinsics Guide



Intrinsics

`__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++

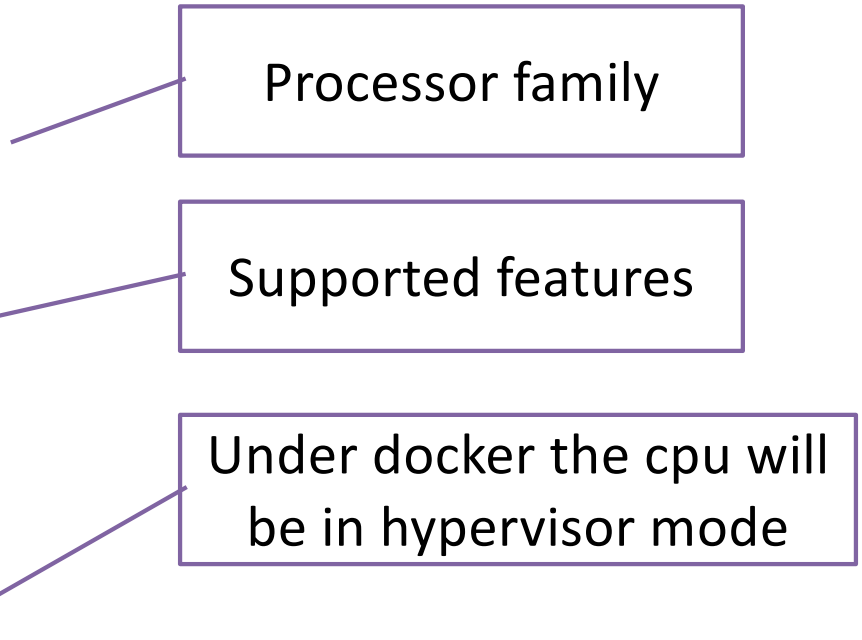
The machine instruction(s) that is/are generated

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
```



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

```
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!!

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

Without Inlining

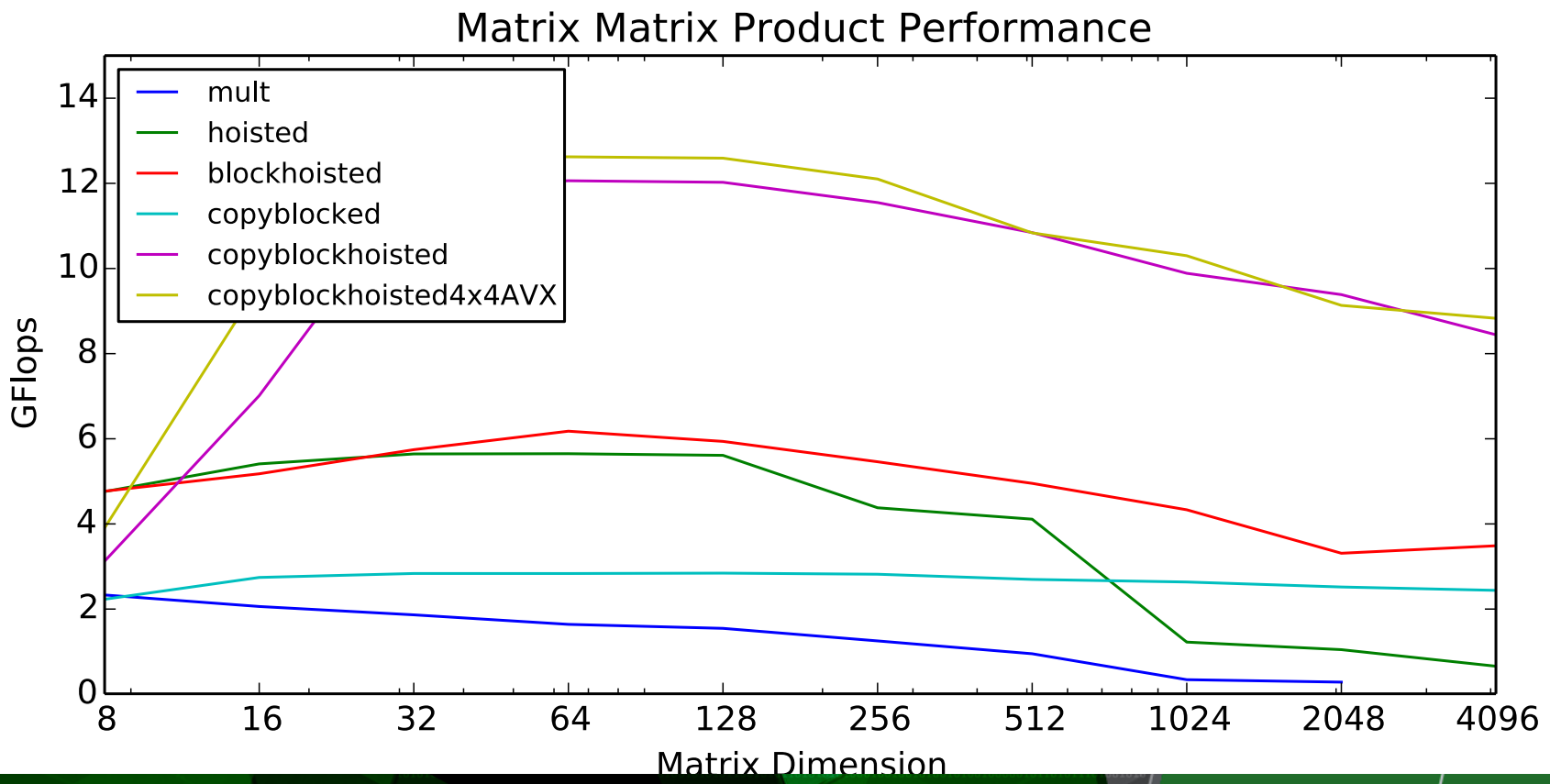
operator()() function call

operator()() function call

operator()() function call

```
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)
```

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

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.

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 $\|\mathrm{Re}(x_i)\| + \|\mathrm{Im}(x_i)\|$ 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 $n \times n$

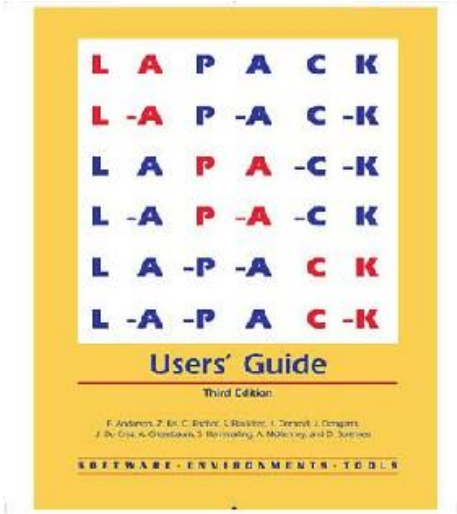
name	description	equation	prefixes
_ger	general rank-1 update	$A = A + \alpha x y^T$	s, d
_geru	(complex)	$A = A + \alpha x y^T$	c, z
_gerc	(complex conj)	$A = A + \alpha x y^H$	c, z
_her	hermetian rank-1 update	$A = A + \alpha x x^H$	c, z
_hpr	(packed)	$A = A + \alpha x x^H$	c, z
_her2	hermetian rank-2 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 $B = \alpha B A^*$	s, d, c, z
_trsm	triangular solve mat	$B = \alpha A^{-*} B$ or $B = \alpha B A^{-*}$	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.

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.

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlops}}{\text{second}} = \min \begin{cases} \text{CPU Peak} \\ \text{Bandwidth} \end{cases}$$

Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4 (2009): 65-76.

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlops}}{\text{second}} = \min \begin{cases} \text{CPU Peak} \ \frac{\text{GFlops}}{\text{second}} \\ \text{Bandwidth} \ \frac{\text{Gbytes}}{\text{second}} \end{cases}$$

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlops}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlops}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \frac{\text{GFlops}}{\text{Gbyte}} \end{cases}$$

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

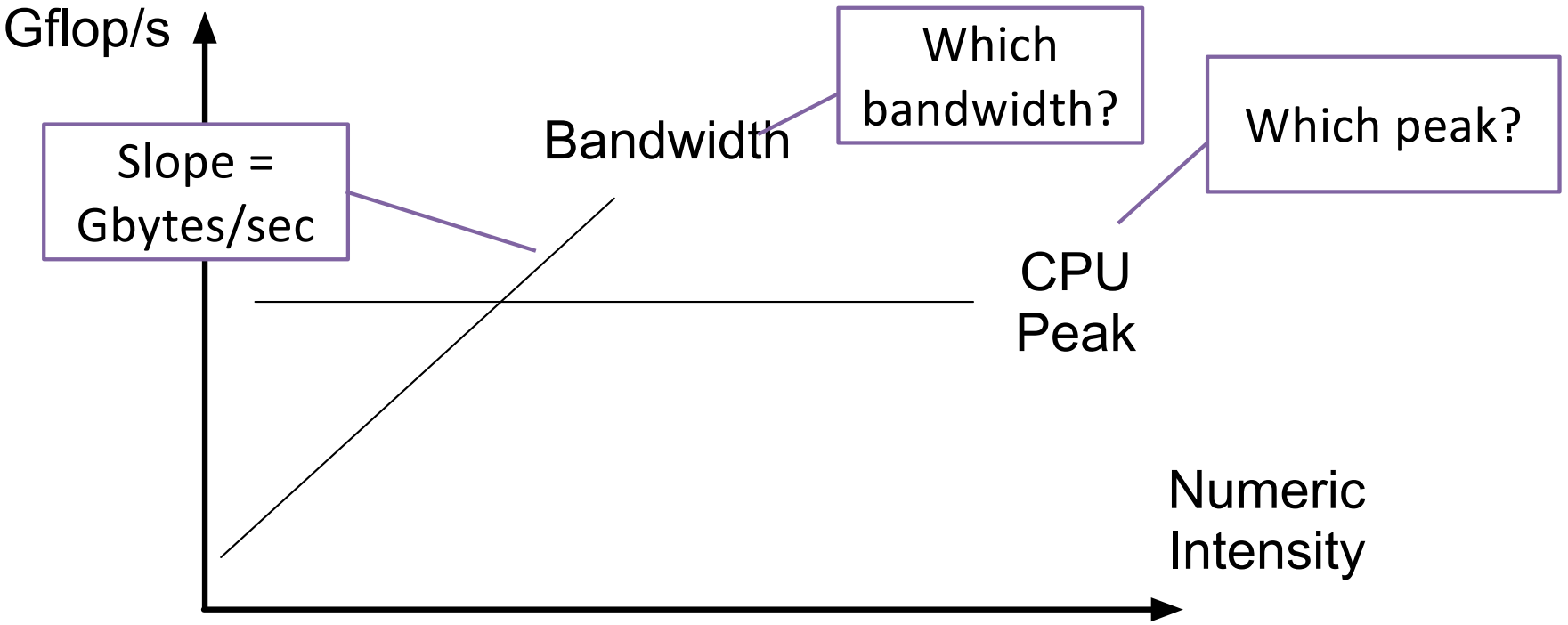
Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

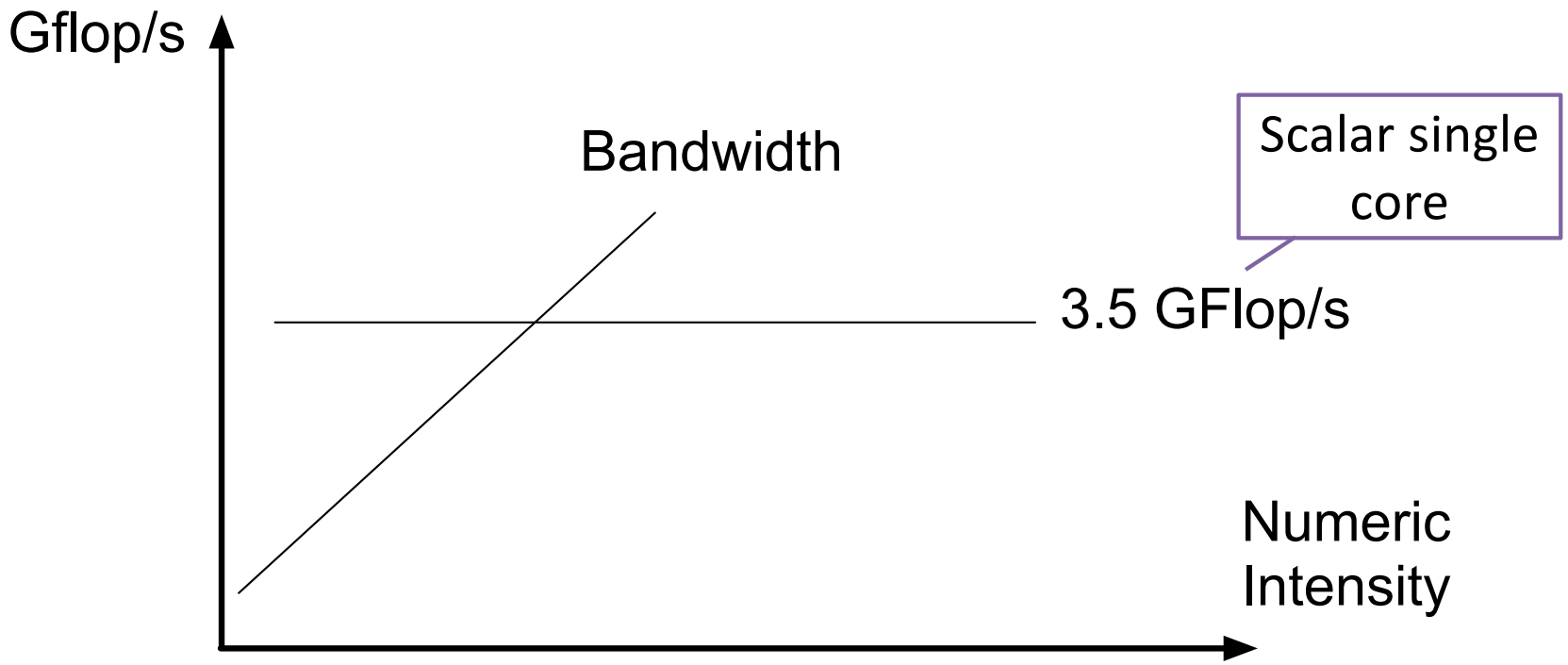
$$\text{Performance} = \frac{\text{GFlops}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlops}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \text{Numerical Intensity } \frac{\text{GFlops}}{\text{Gbyte}} \end{cases}$$

Roofline Model

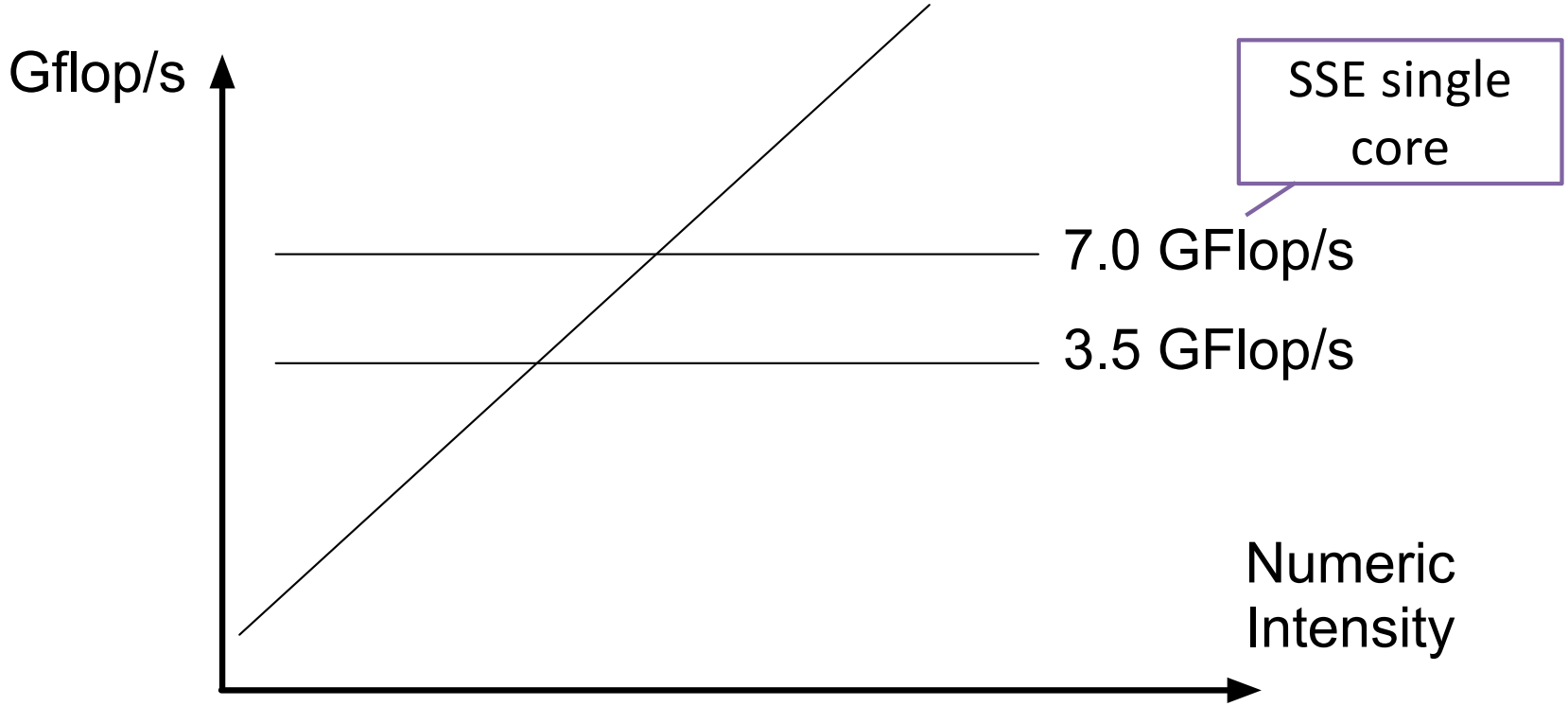
$$\text{Performance} = \frac{\text{GFlops}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlops}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \text{Numerical Intensity } \frac{\text{GFlops}}{\text{Gbyte}} \end{cases}$$



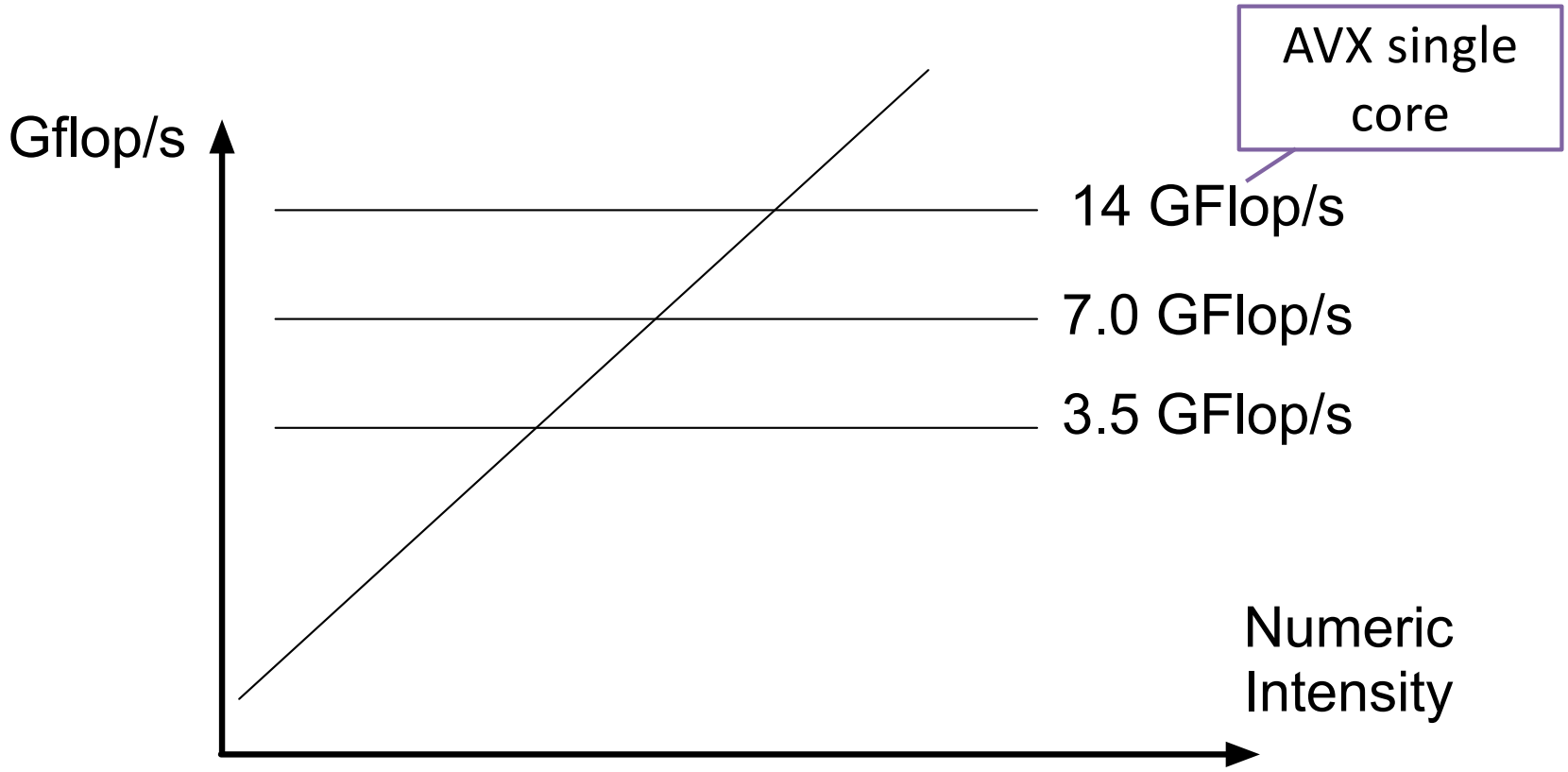
Roofline Model



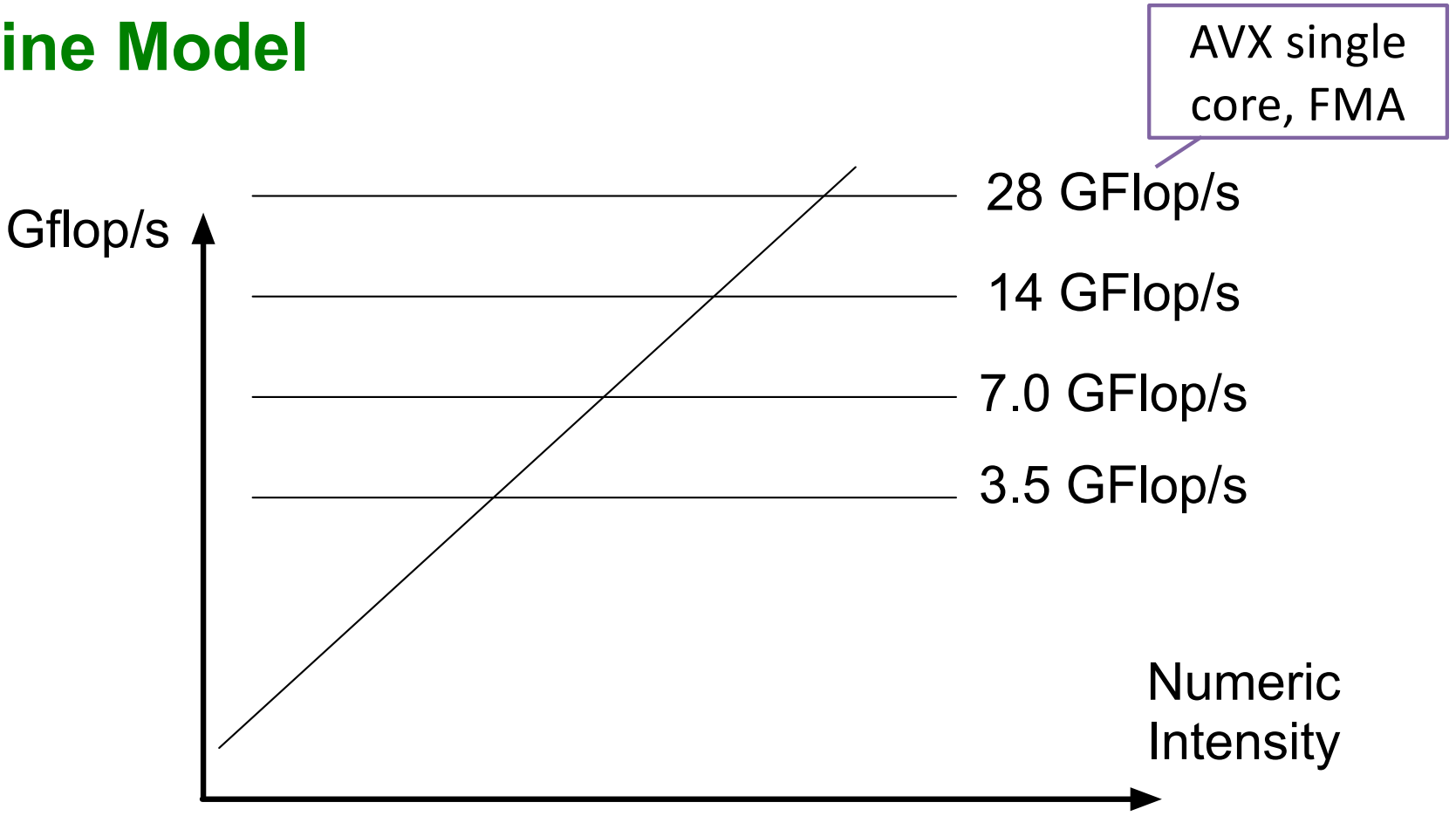
Roofline Model



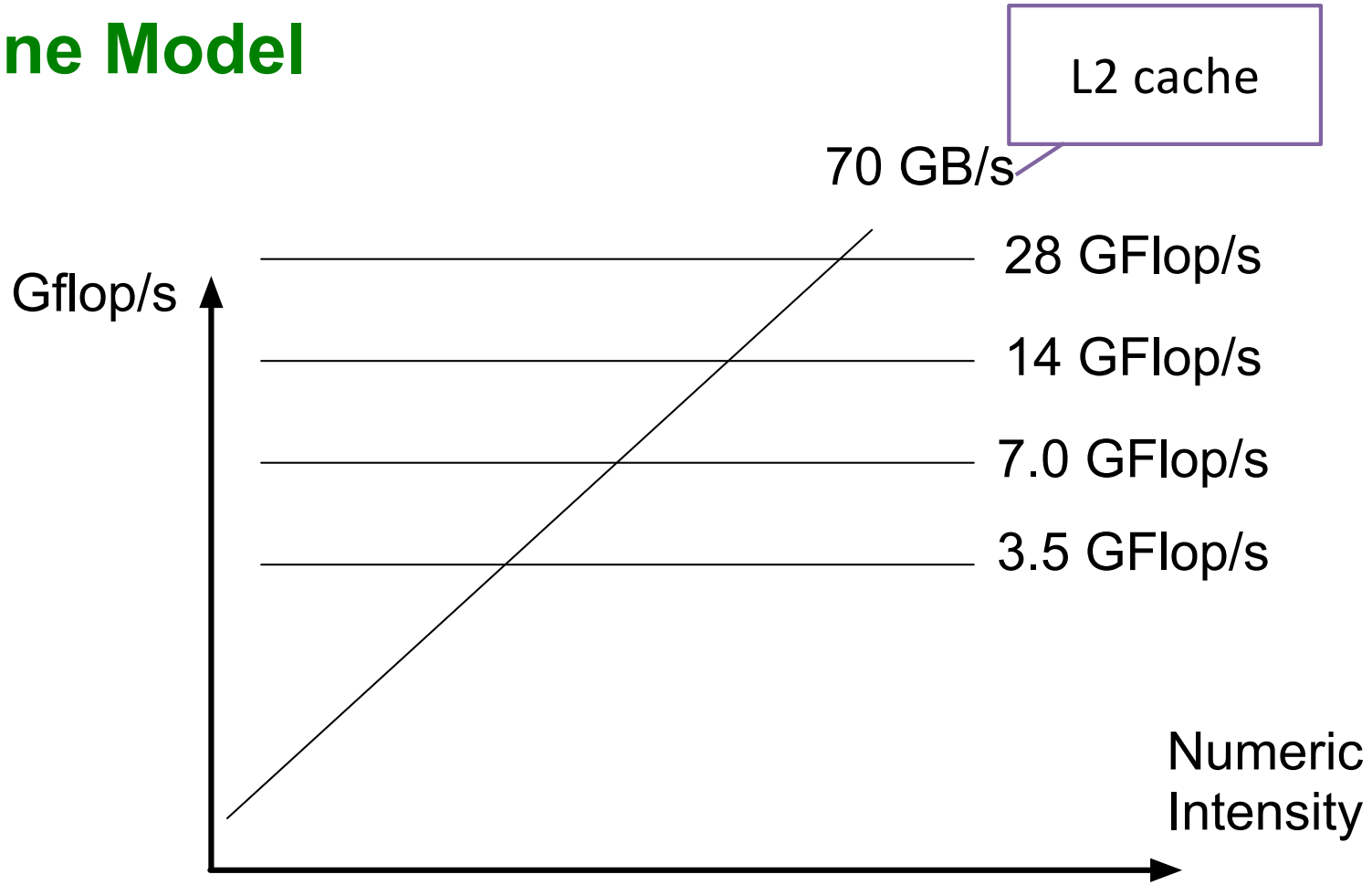
Roofline Model



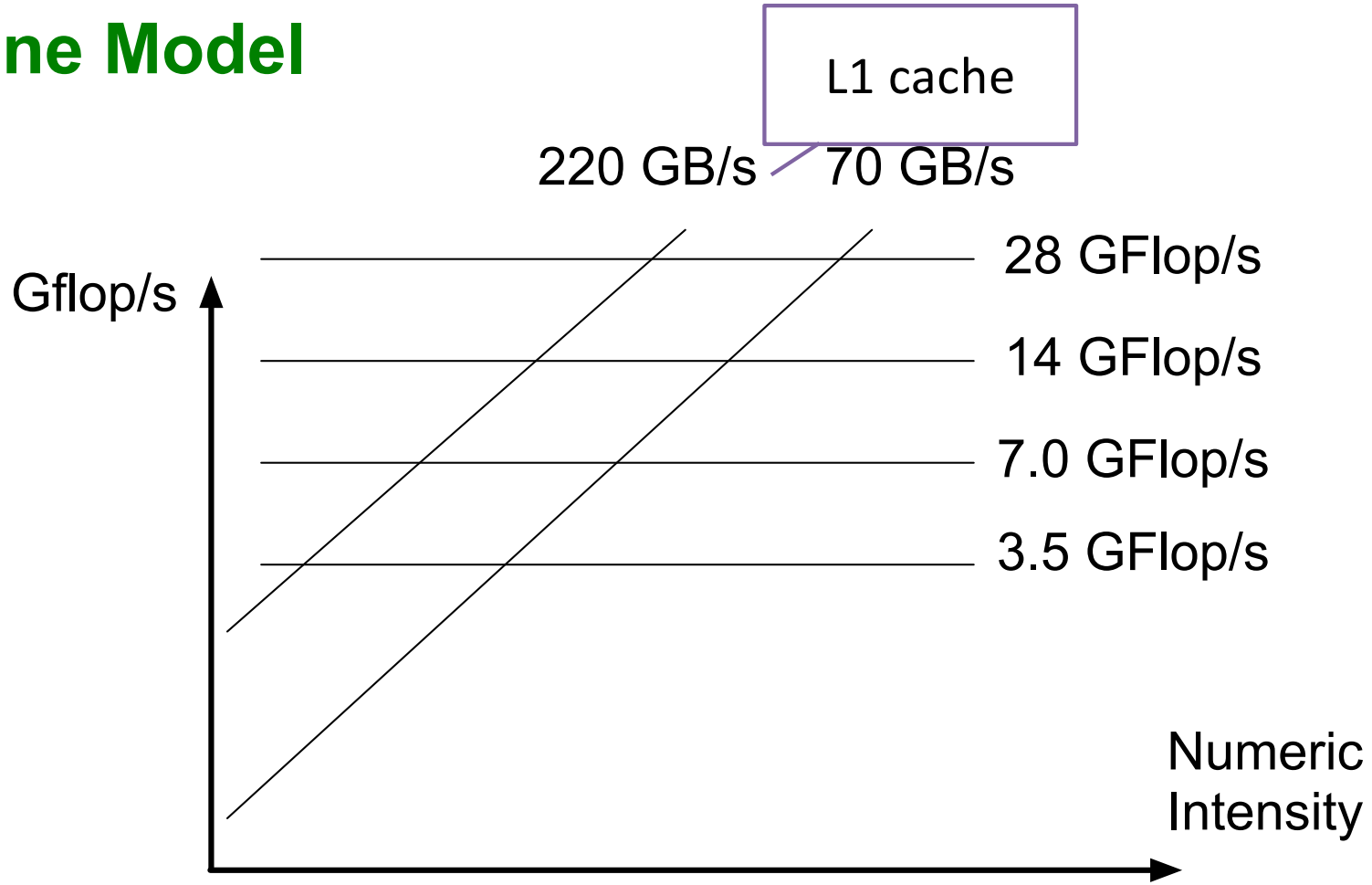
Roofline Model



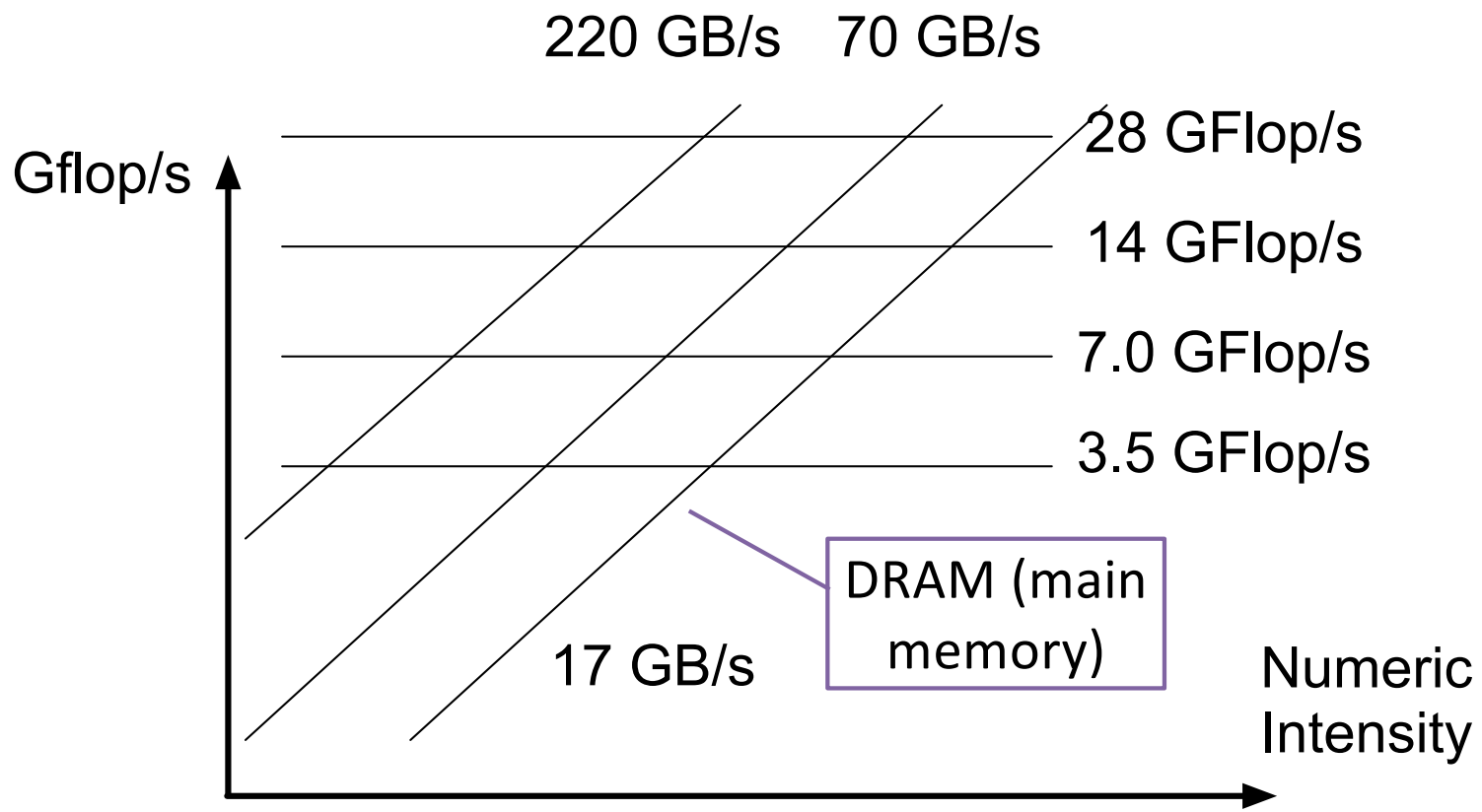
Roofline Model



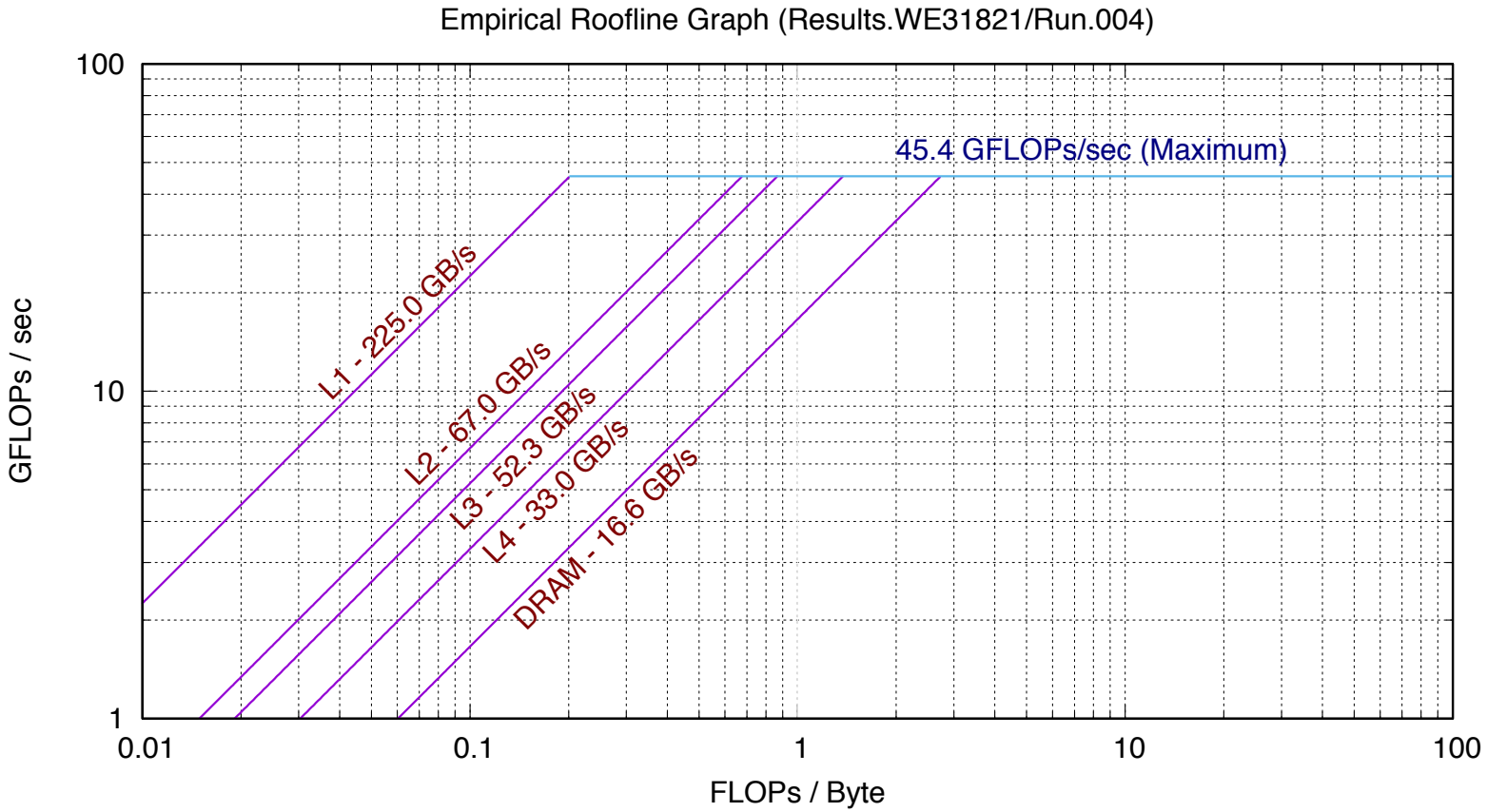
Roofline Model



Roofline Model



Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
GFLOPs / sec
FLOPs / Byte
100
10
1
0.01
0.1
1
10
100

Numerical Intensity

```
void matvec_dense(const Matrix& A, const Vector& x, Vector& y) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < A.num_cols(); ++j) {
      y(i) += A(i, j) * x(j);
    }
  }
}
```

Three doubles = 24 bytes?

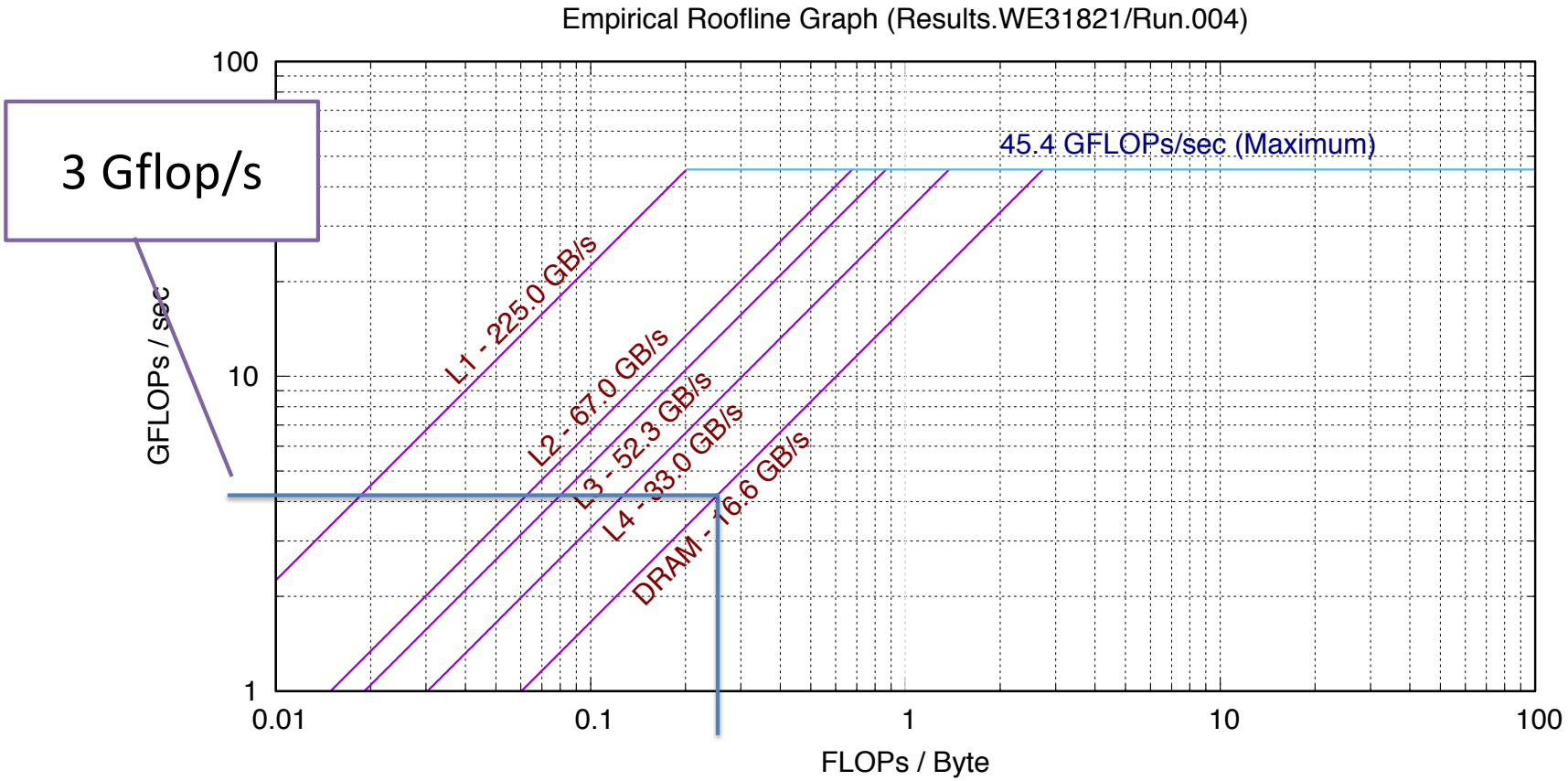
Two flops

$2N^2$ Flops

$(N^2 + 2N)$ doubles
$= 8(N^2 + 2N)$ bytes

$\frac{1}{4} \frac{\text{Flop}}{\text{byte}}$

Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
3 Gflop/s
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
GFLOPs / sec
FLOPs / Byte
100
10
1
0.01
0.1
1
10
100

Numerical Intensity

```
void matvec(const Vector& x, Vector& y) const {
  for (size_type k = 0; k < arrayData.size(); ++k) {
    y(rowIndices[k]) += arrayData[k] * x(rowIndices[k]);
  }
}
```

Three doubles + 2 ints
= 32 bytes? (36 bytes?)

Two flops

5N

$2\,\text{NNZ}$ Flops

10N Flops

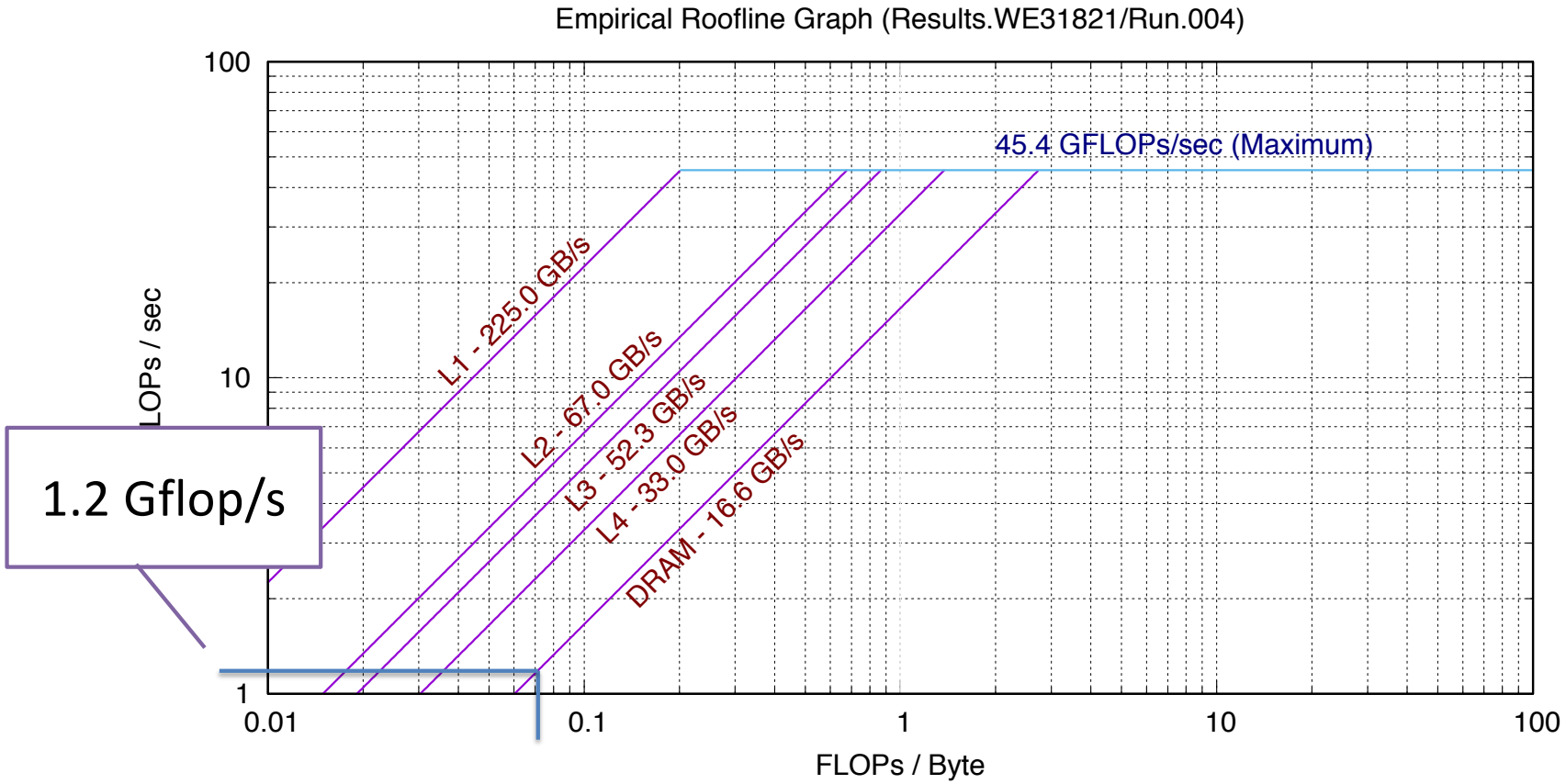
7N doubles = 56 bytes

NNZ doubles
$+2\,\text{NNZ}$ indexes
$+2N$ doubles

$\frac{1}{14}\,\frac{\text{Flop}}{\text{byte}}$

10N indexes = 40, 80 bytes

Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
100
10
1
0.01
0.1
1
10
100
LOPs / sec
FLOPs / Byte
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
1.2 Gflop/s

Numerical Intensity

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

Three doubles = 24 bytes?

Two flops

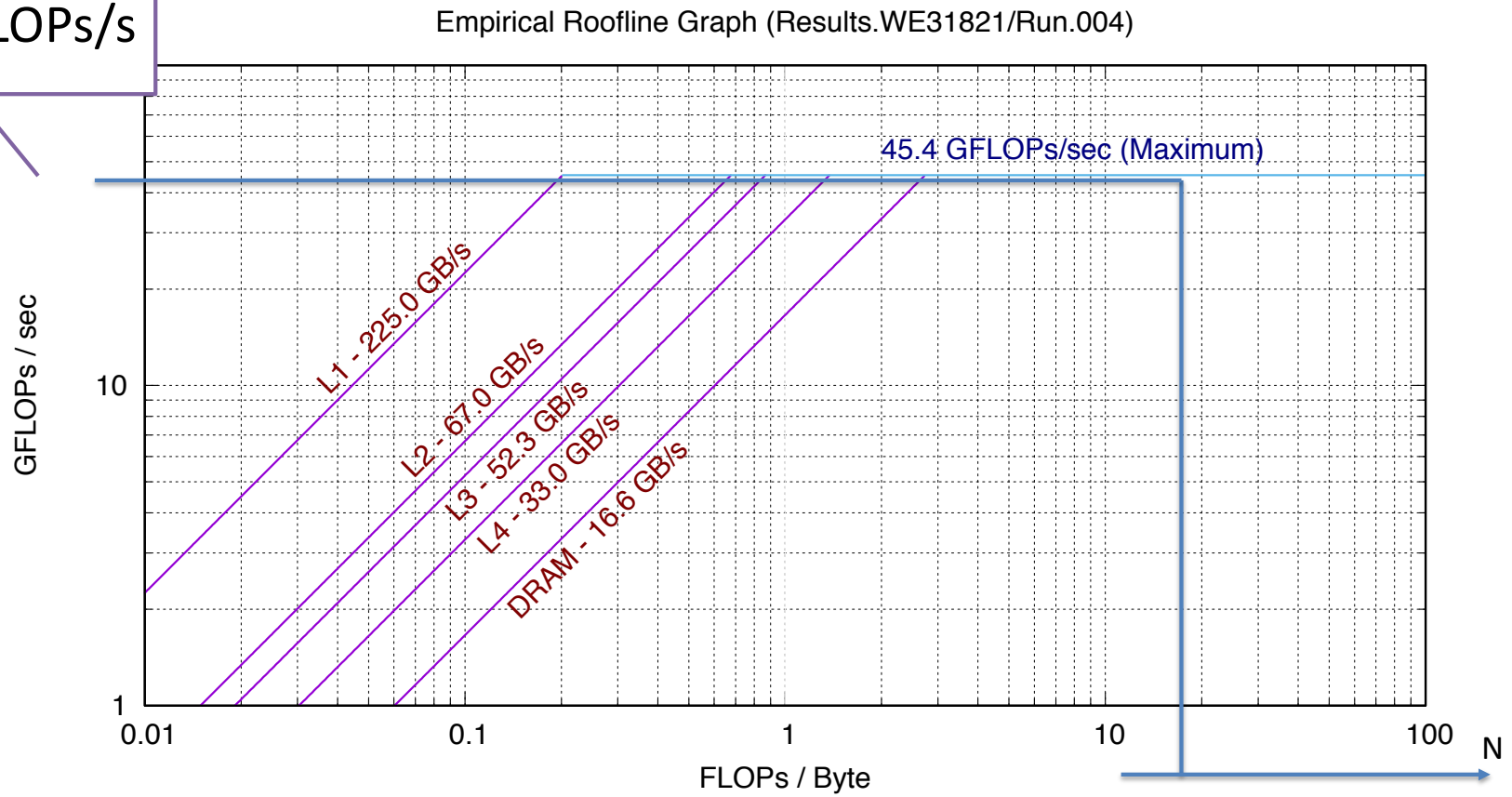
$\frac{N}{12} \frac{\text{Flop}}{\text{byte}}$

$2N^3$ Flops

$3N^2$ doubles $= 24N^2$ bytes

Measured

45 GFLOPs/s



Numerical Intensity

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

Three doubles = 24 bytes?

Two flops

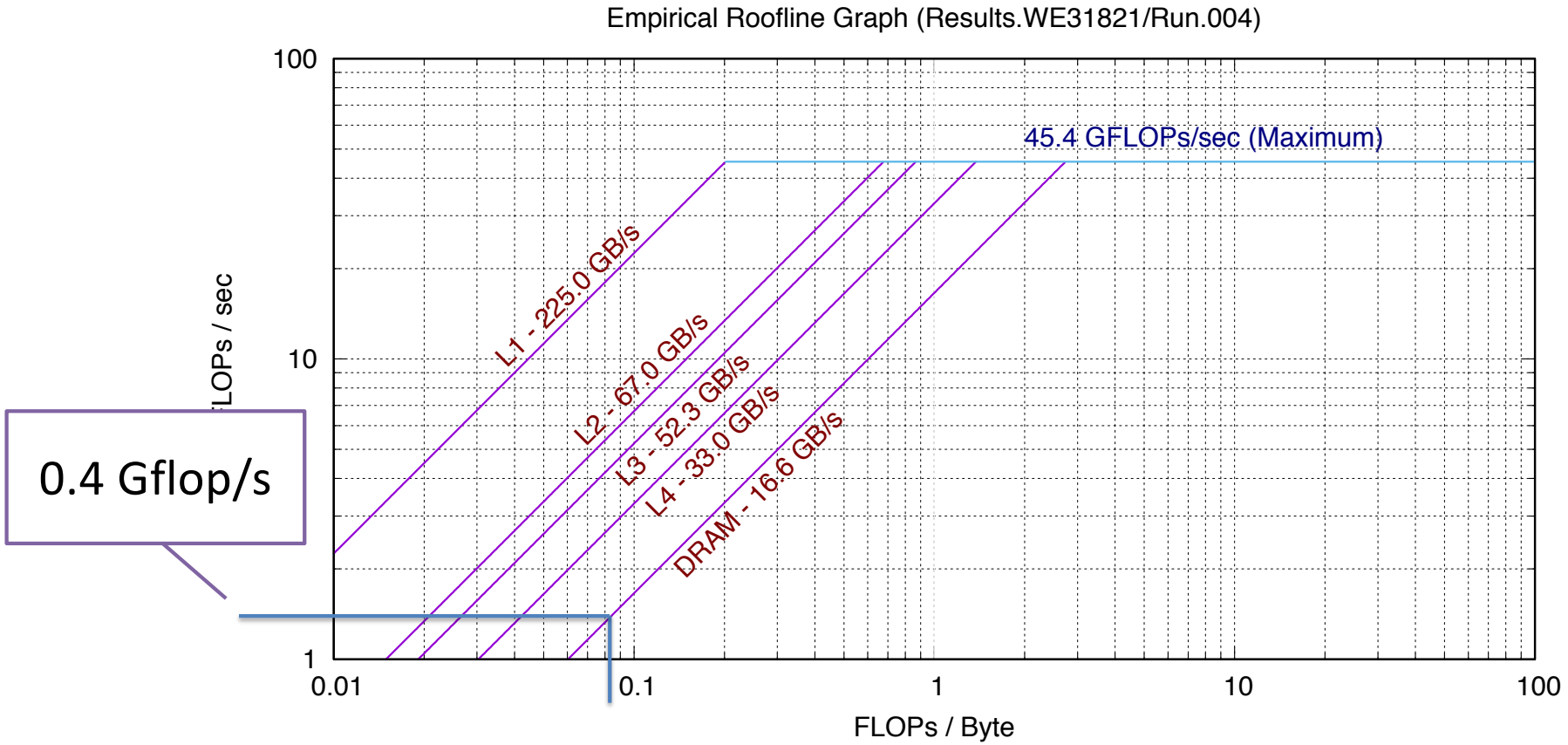
$\frac{1}{12} \frac{\text{Flop}}{\text{byte}}$

$2N^3$ Flops

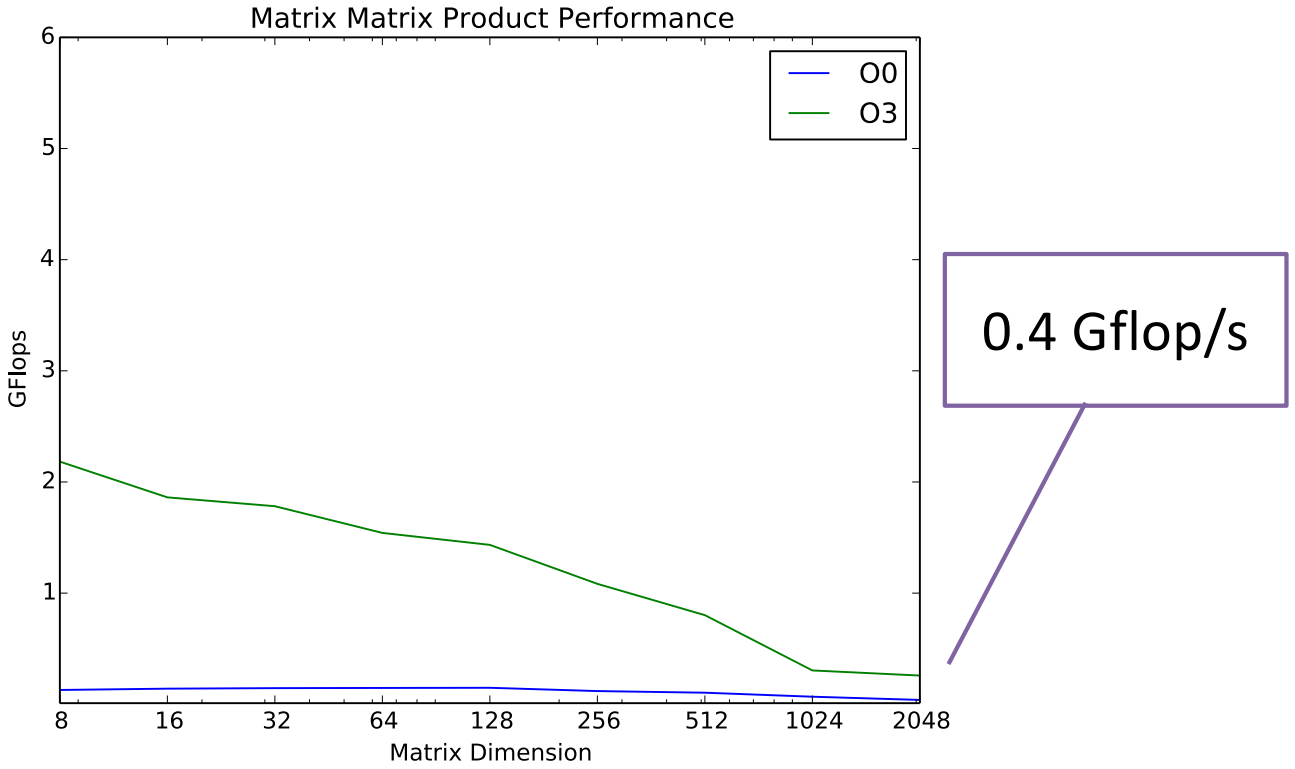
$3N^3$ doubles $= 24N^3$ bytes

No reuse

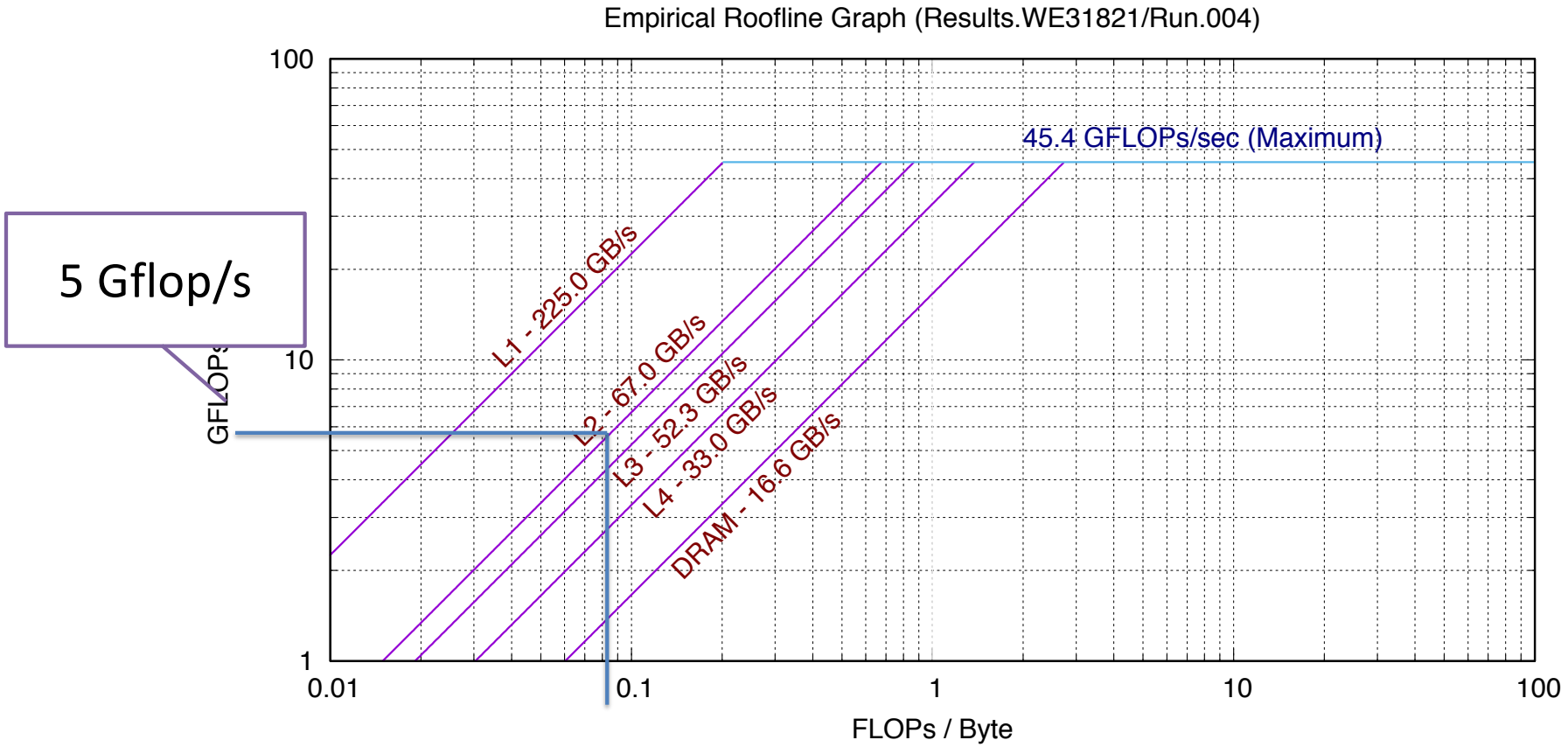
Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
100
10
1
0.01
0.1
1
10
100
FLOPs / Byte
LOPs / sec
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
0.4 Gflop/s

Recall


Matrix Matrix Product Performance
GFlops
Matrix Dimension
O0
O3
0
1
2
3
4
5
6
8
16
32
64
128
256
512
1024
2048
0.4 Gflop/s

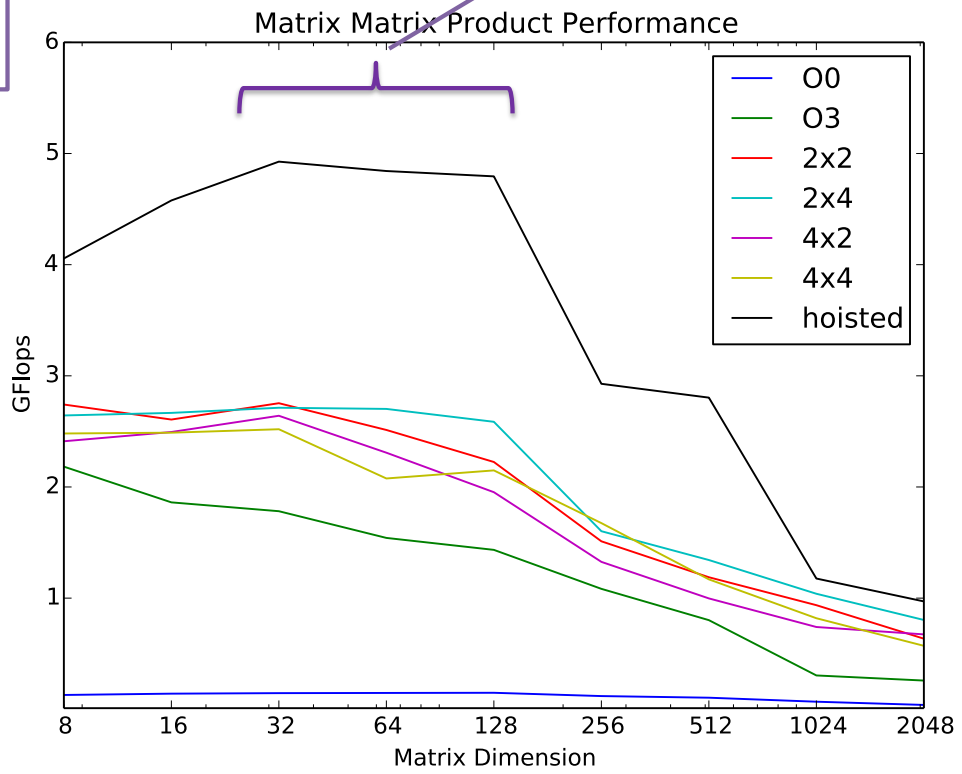
Hoisting / tiling etc: L2



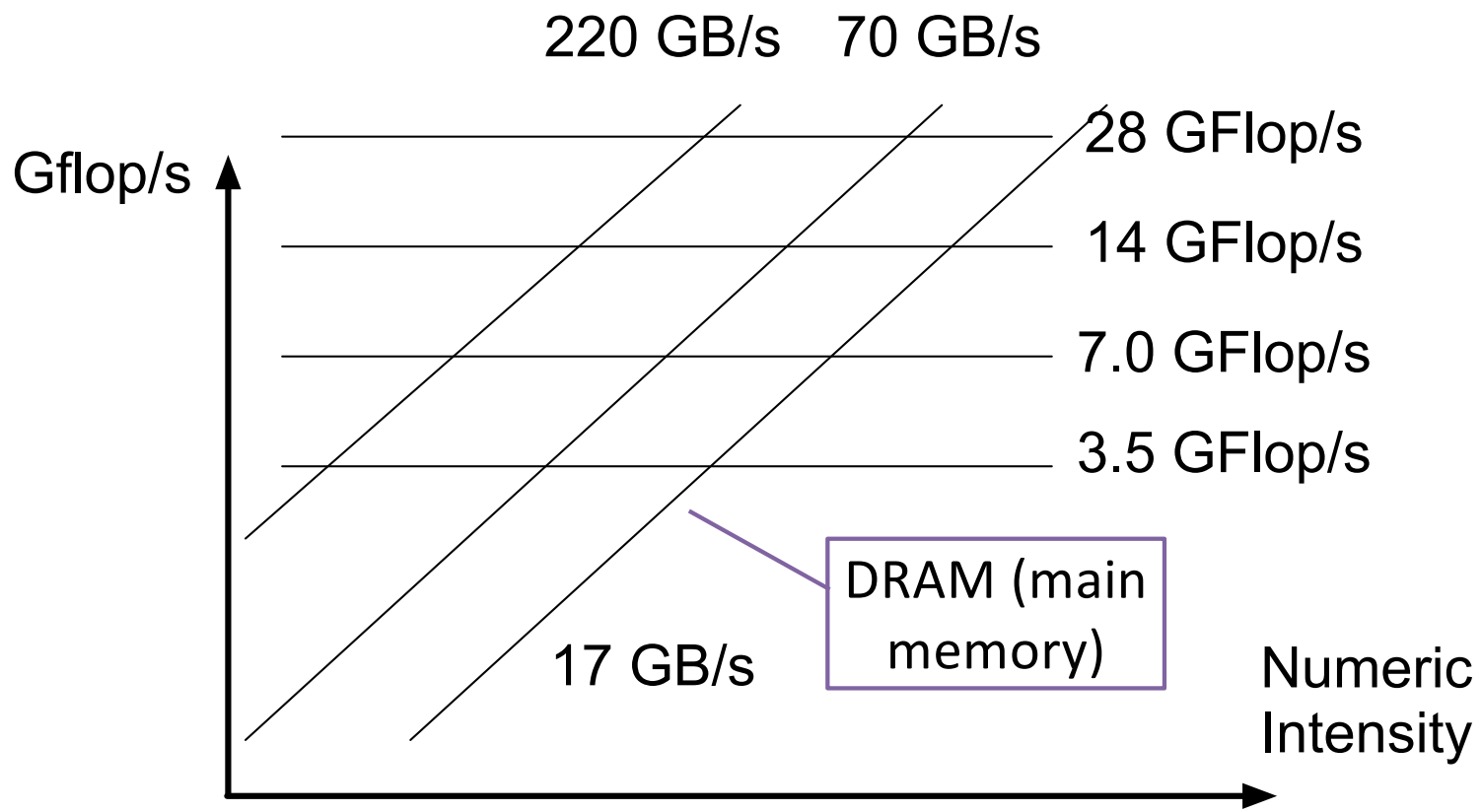
Hoisting / tiling etc: L2

L2 zone

5 Gflop/s



Summary (Roofline Model)



Thank You!

Creative Commons BY-NC-SA 4.0 License