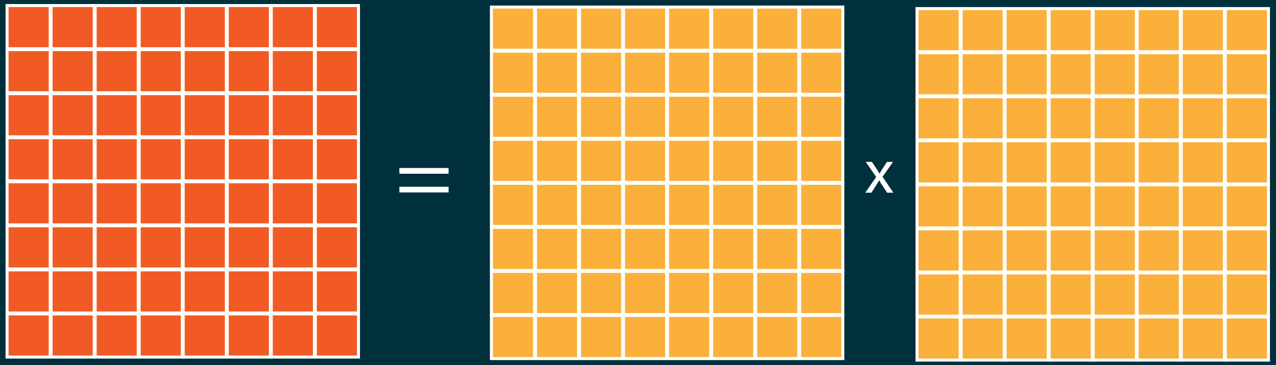


Optimizing Single Core GEMM

CS267 Recitation 1

GSI: Vivek Bharadwaj

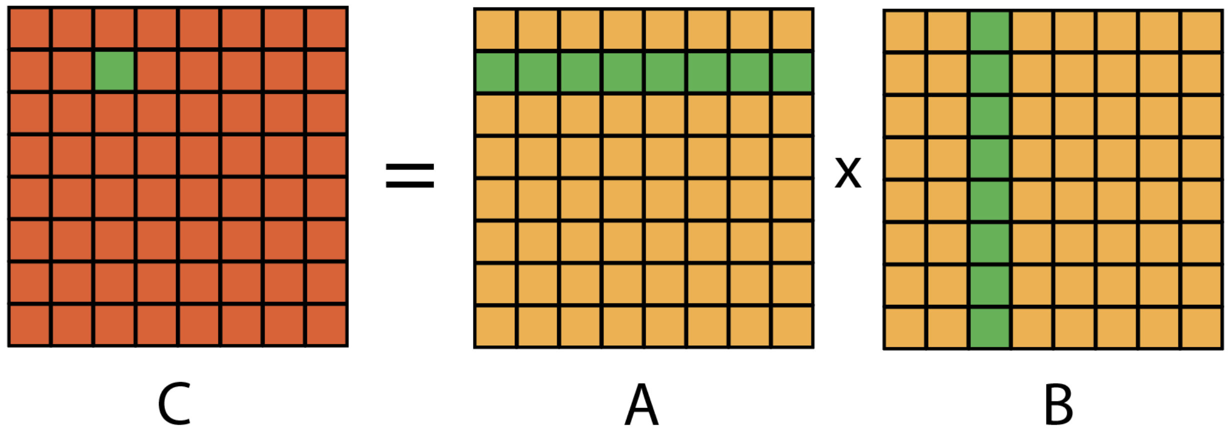
Cori KNL Version

Class Notes

- This recitation should be recorded (please tell me if I’m not recording!)
- By now, you should have gotten access to Cori and set up your NERSC account
- Make sure that you can SSH in; set up SSHProxy if you’re tired of retyping your password every time. Set up a good working environment (Vim + tmux, VSCode are popular)
- **REMEMBER:**
  - Compile on login nodes (or else you’ll wait forever)
  - Run code in batch jobs or interactive nodes (or you’ll slow the login nodes for everyone)
  - For this assignment, use **KNL nodes**, not Haswell

Assignment 1

- Write the **fastest** single core matrix multiplication code for square matrices that you can.



- Let the matrix side length be n. For all $i, j \in [n]$, the mathematical formula is

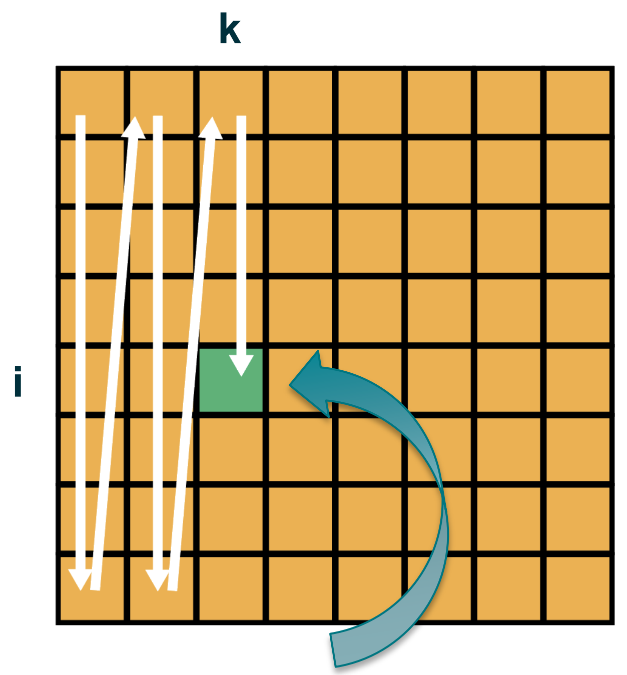
$$\mathbf{C}[\mathbf{i},\mathbf{j}] = \mathbf{A}[\mathbf{i},:] \cdot \mathbf{B}[:,\mathbf{j}]$$

- Read as: element (i, j) of C is a dot product between row i of A and column j of B
- Solution involves many fundamental high-performance computing (HPC) techniques

The Simple Version

- All three of your matrices are all stored in **column-major** order, so elements in each column are adjacent in each memory block (a long 1D array)
- The simplest possible code is:

```
void simpleGEMM(double *A, double *B, double* C, int n) {
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      C[i + j * n] = 0.0;
      for(int k = 0; k < n; k++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
      }
  ...
}
```



The (i, k) entry is stored at A[i + k * n]

Grading and Report

- Your job: optimize the simple code on the previous slide for a SINGLE core (no OpenMP or multicore parallelism)

- Lots of things you can try. There's a whole body of literature on matrix multiplication. See the assignment page for links, do research, try interesting ideas that you find.

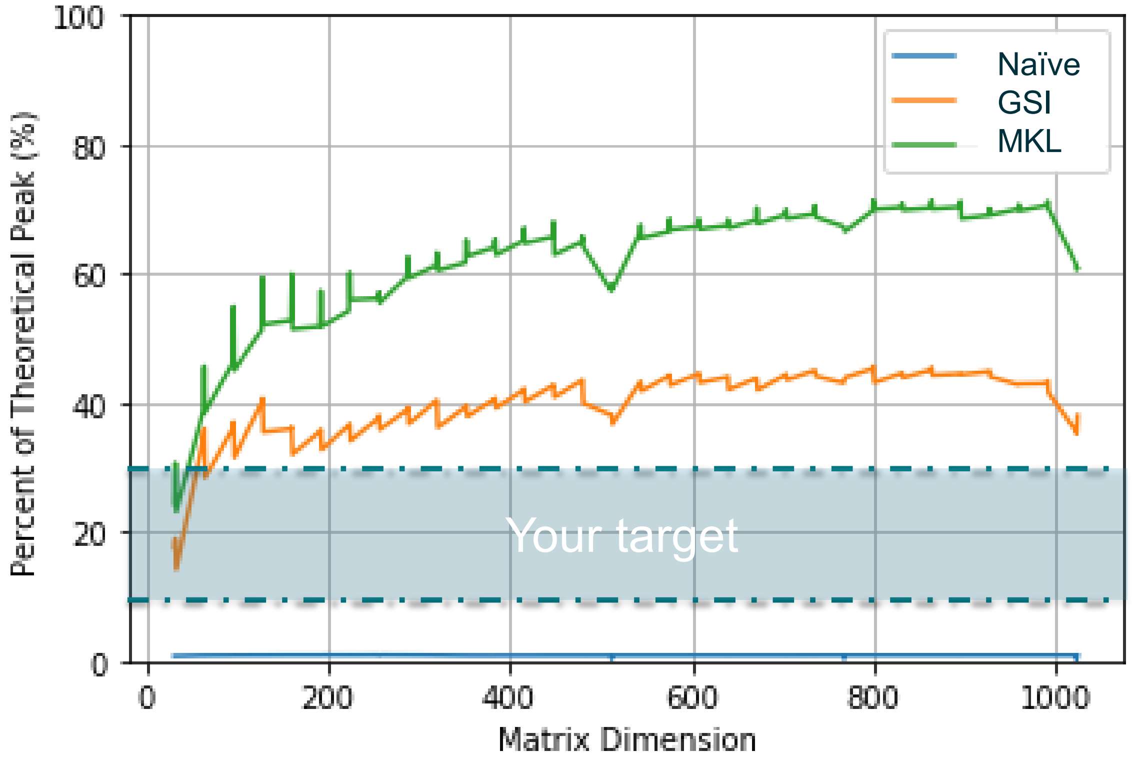
- You are graded on:
  - Percentage of theoretical machine peak (relative to the entire class's performance)
  - Your report, which should include:
    - Techniques you tried and relation to previous matrix multiplication work
    - Justification for your techniques (any formulae or calculation you used in designing your algorithm), diagrams, pseudocode, design choices
    - Benchmarks, results of searches, and performance graphs

Performance Target

- Last year's record: 40% of the peak (averaged over several values of n), still lots of room for improvement

- Goal: at least 10-30% of the peak

- Optimizations are more than additive. For example, writing a micro-kernel and using SIMD might each produce a small improvement…
  – But together (done correctly), they produce significant improvement



Optimizations for Single Core GEMM

Sorted roughly by difficulty. DO NOT need to do all of them, don't have to follow this order, can try ANY other technique. These are just some ideas; do what you can.

01

Single level of blocking (already in the starter code)

02

Multiple Levels of Blocking

03

Repack and realign matrices

(AKA copy optimizations)

04

Optimize loop order for ILP

05

Microkernels!

06

SIMD Microkernels

07

Software prefetching

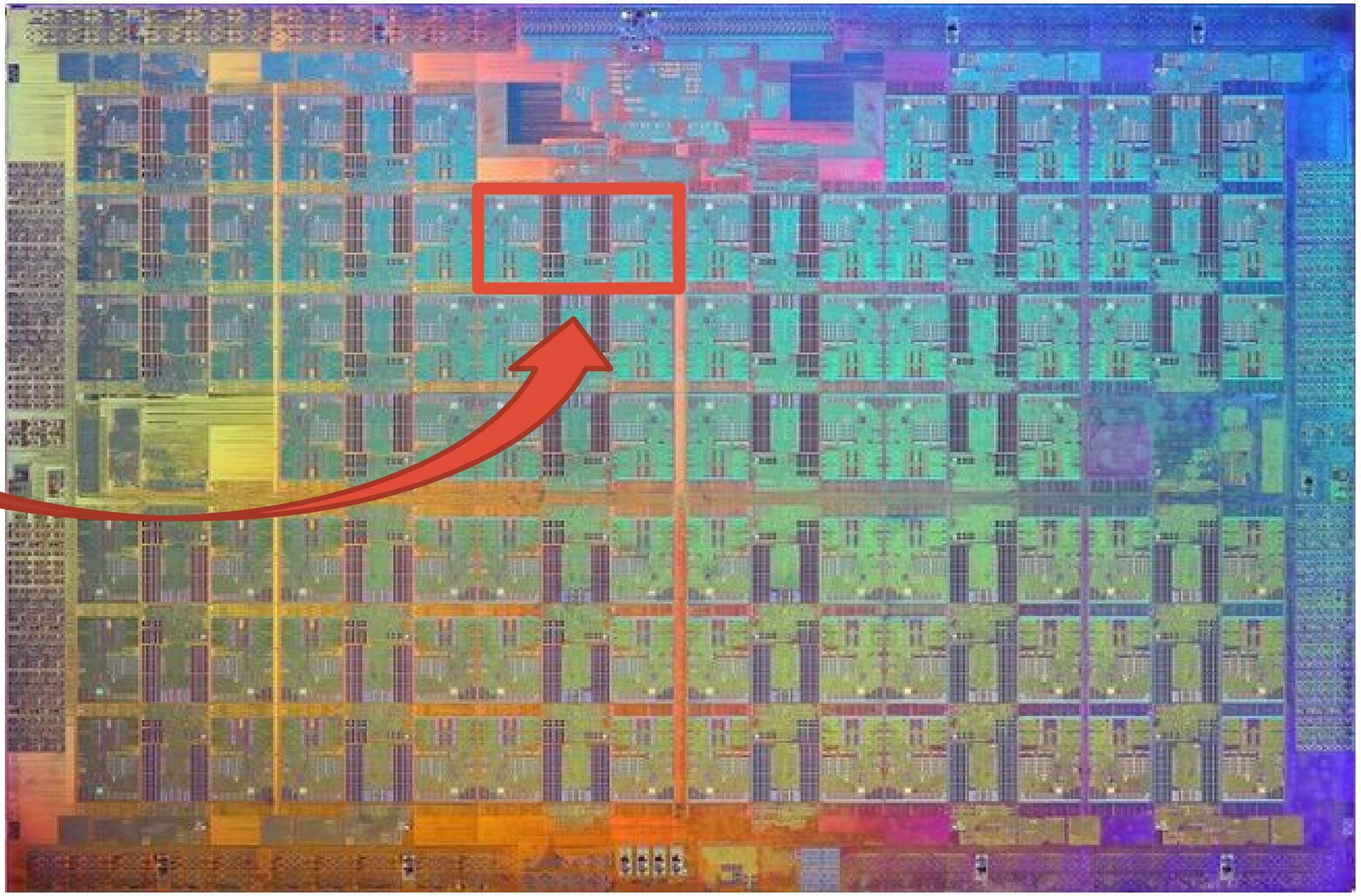
08

Write inline assembly to take advantage of embedded broadcast

Meet the Processor: Intel Xeon Phi

Our model: “Knights Landing”

Two Adjacent Cores



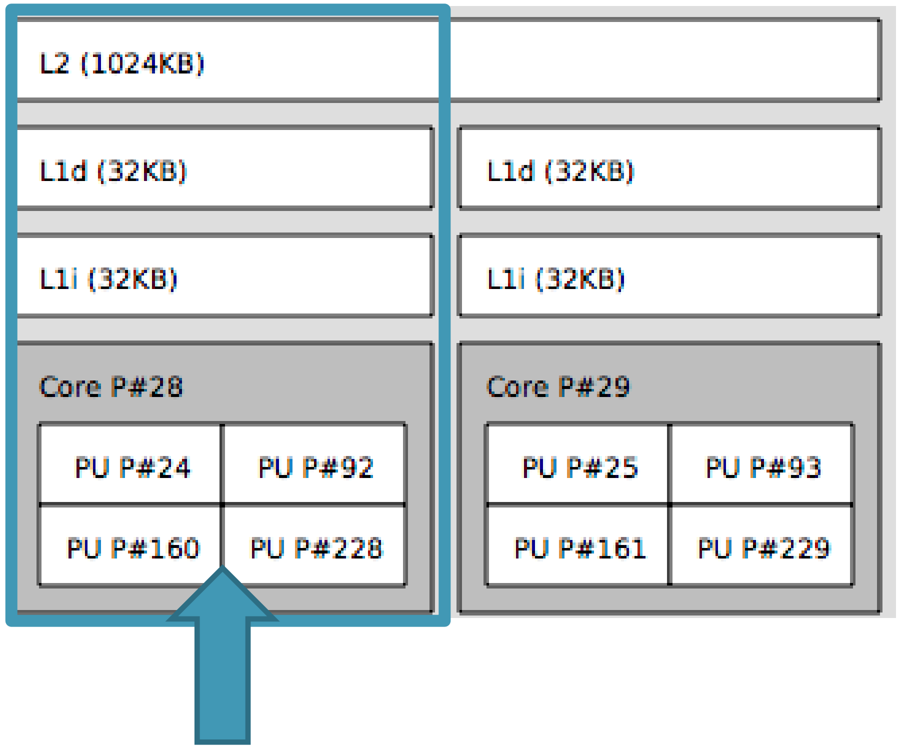
*Not every core is enabled (any guesses why?)

Meet the Processor: Intel Xeon Phi

- 68 cores per node (but you will only use 1)
- Each core:
  - Runs at 1.4 GHz
  - Supports AVX512 vector operations that operate on 8 double precision floats with one instruction
  - Two vector lanes (can process 2 vector instructions simultaneously)
  - Maximum Possible Throughput (FMADD -> 2 FLOPS)

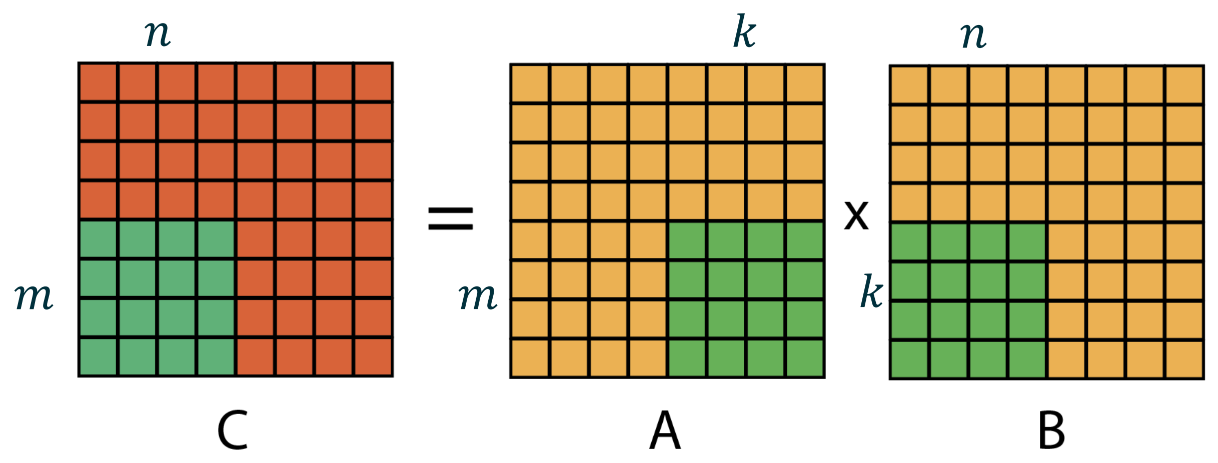
$$1.4 \times 10^9 \frac{\text{instructions}}{\text{lane} \cdot s} \times 16 \frac{\text{FLOP}}{\text{instruction}} \times 2 \text{ lanes} = 44.8 \text{ GFLOPs}$$

  - Can we hit the peak? Nope – processor is starved by memory bandwidth. **USE THE CACHES / REGISTERS!**
    - L2 Cache: 1 MB
    - L1 Data Cache: 32 KB
    - Registers: 32 AVX512 vector registers holding 8 doubles each



Each PU is a hyperthread; you can ignore this.

Idea 1: Single Level of Blocking



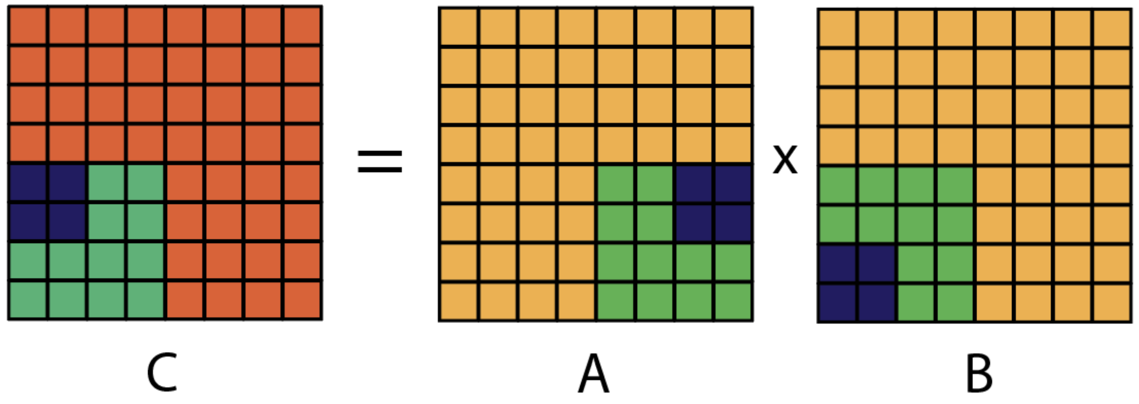
- Break the large problem into a series of smaller matrix multiplications (multiplication of $m \times k, k \times n$ blocks). See the lecture slides for why this helps cache efficiency.
- Choose a cache level to target (L2, L1) and compute the total number of data words C that it can hold. May want to use a value of C lower than the theoretical maximum.
- Choose block tiling parameters m, n, k such that
$$mn + mk + kn \leq C$$
- Code already provided in the starter. You should **search over different tile shapes**, report findings. **Starter code will not achieve high performance without further modification.**

Starter Code

```c
void square_dgemm(int lda, double* A, double* B, double* C) {
    // For each block-row of A
    for (int i = 0; i < lda; i += BLOCK_SIZE) {
        // For each block-column of B
        for (int j = 0; j < lda; j += BLOCK_SIZE) {
            // Accumulate block dgemms into block of C
            for (int k = 0; k < lda; k += BLOCK_SIZE) {
                int M = min(BLOCK_SIZE, lda - i);
                int N = min(BLOCK_SIZE, lda - j);
                int K = min(BLOCK_SIZE, lda - k);
                // Perform individual block dgemm
                do_block(lda, M, N, K,
                        A + i + k * lda,
                        B + k + j * lda,
                        C + i + j * lda);
            }
        }
    }
}
```

```c
static void do_block(int lda, int M, int N, int K, double* A,
double* B, double* C) {
    // For each row i of A
    for (int i = 0; i < M; ++i) {
        // For each column j of B
        for (int j = 0; j < N; ++j) {
            // Compute C(i,j)
            double cij = C[i + j * lda];
            for (int k = 0; k < K; ++k) {
                cij += A[i + k * lda] * B[k + j * lda];
            }
            C[i + j * lda] = cij;
        }
    }
}
```

Idea 2: Multiple Levels of Blocking

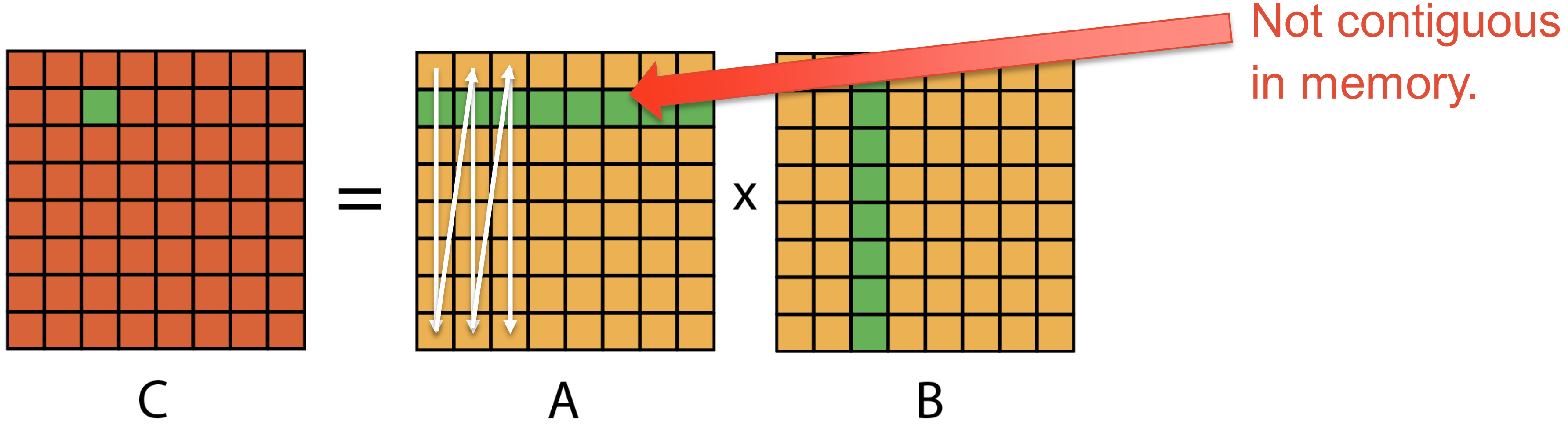


- Recurse! Perform additional nested levels of blocking. Use the starter code for an example. If you targeted L2 for the first level, target either L1 or registers for the second level. Benchmark and make graphs for different tile shapes, etc.

- Same **nesting order** of three loops *may not be optimal* for all three layers! Try changing it (three nested loops remain correct in any order; read GotoBLAS paper for optimality of orders)

- A subprocedure for GEMM that is blocked for registers is typically called a **micro-kernel**

Idea 3: Repack and Realign Your Matrices



- Assume that you aren’t doing any blocking and you’re calling the naïve GEMM code. Each dot product requires a row of *A*, column of *B*.
- Accesses to each element of *A* are separated by *n* words in memory; not contiguous.
- Non-contiguous memory accesses can’t take advantage of hardware prefetching, run slower.

Idea 3: Repack and Realign Your Matrices

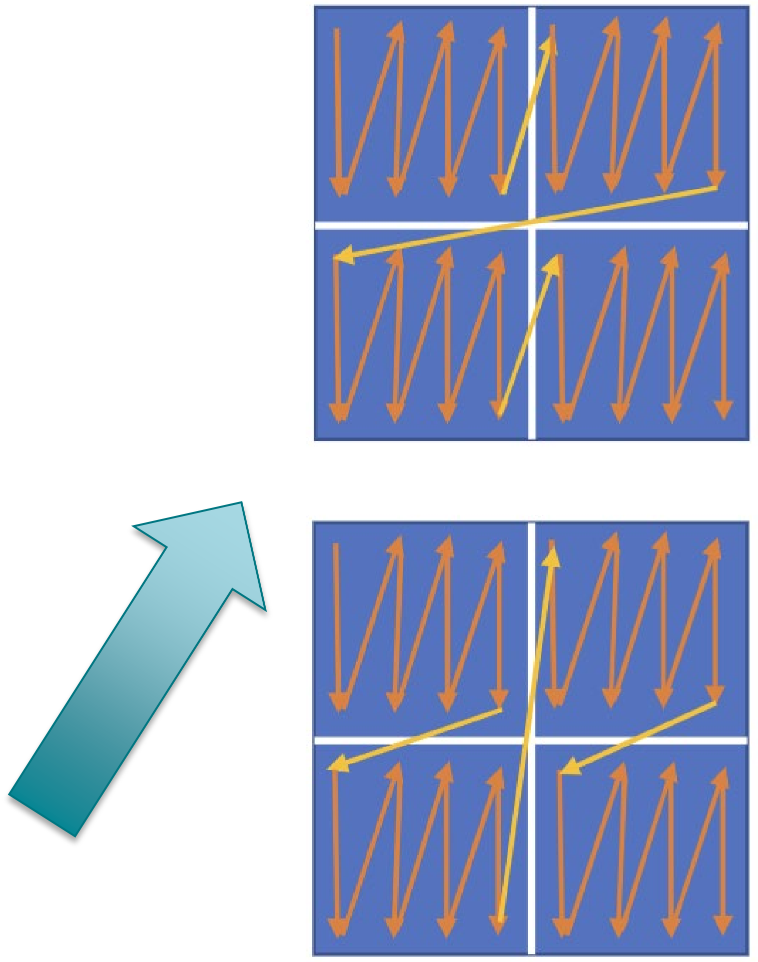
- **Solution for the naïve code: copy** over A to a new memory location and repack so that it's in row-major order before you perform the matrix multiplication.

- Introduces overhead of copy + reorganize. But these cost $O(n^2)$ time while matrix multiplication is $O(n^3)$, so benefits outweigh costs if done correctly.

- Depending on loop order, may want to repack matrix B in column-major; tailor repacking to your algorithm.

- **With blocking: same logic but could keep elements within each block contiguous.**

Blocked column-major orderings.

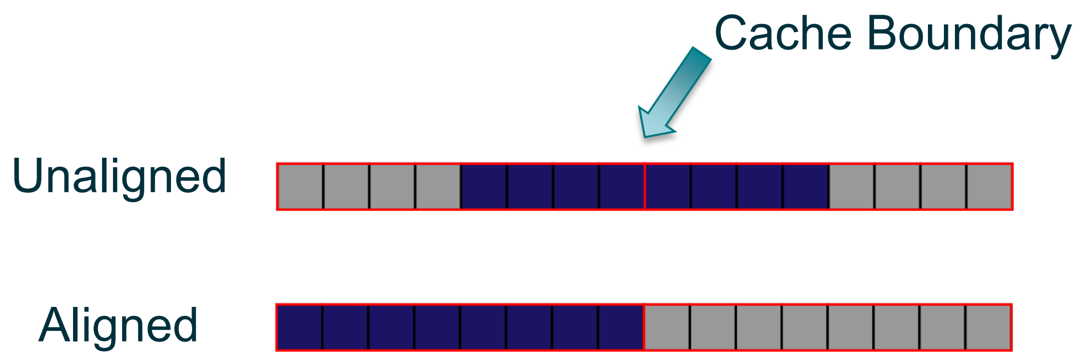


Idea 3: Repack and Realign Your Matrices

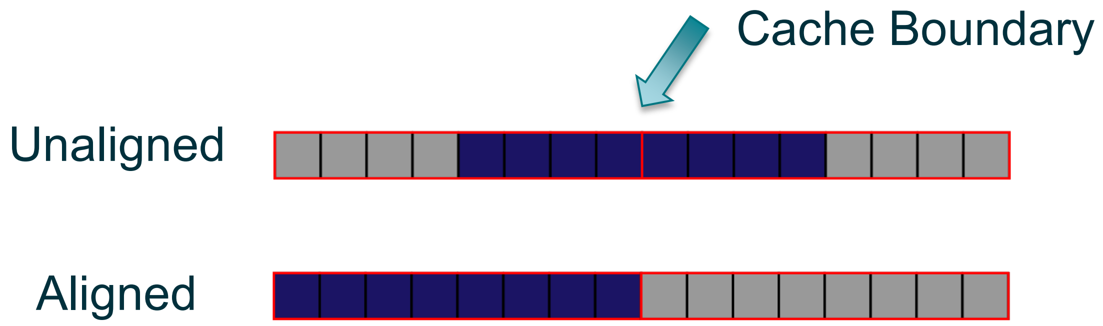
- When you copy over matrices to repack, good idea to **align them to the cache boundary**. May yield a fractional speedup.

- What does that mean? When you call `malloc`, the beginning of your new block of memory is some arbitrary location.

- Basic units in a cache: **blocks** or **lines.** Small group of adjacent memory locations. Each line is 64 bytes on KNL. Line boundaries are always multiples of 8 double words.
  - Unaligned: Start of block is not a multiple of 8
  - Aligned: Start of block is a multiple of 8



Idea 3: Repack and Realign Your Matrices



- Why might you want your new memory block aligned? Certain low-level machine instructions (e.g. SIMD instructions) work on contiguous segments of 8 double words in memory.

- They run faster if you guarantee that the start of the segment is aligned with a cache boundary (i.e. either all elements are in the cache, or all need to be loaded in, nothing “partial”)

- **How to do it?** When copying, instead of `malloc`, call `_mm_malloc(<bytes you want>, 64)` to align to 64-byte boundary. To get the function, need to `#include <immintrin.h>`

Idea 3: Repack and Realign Your Matrices

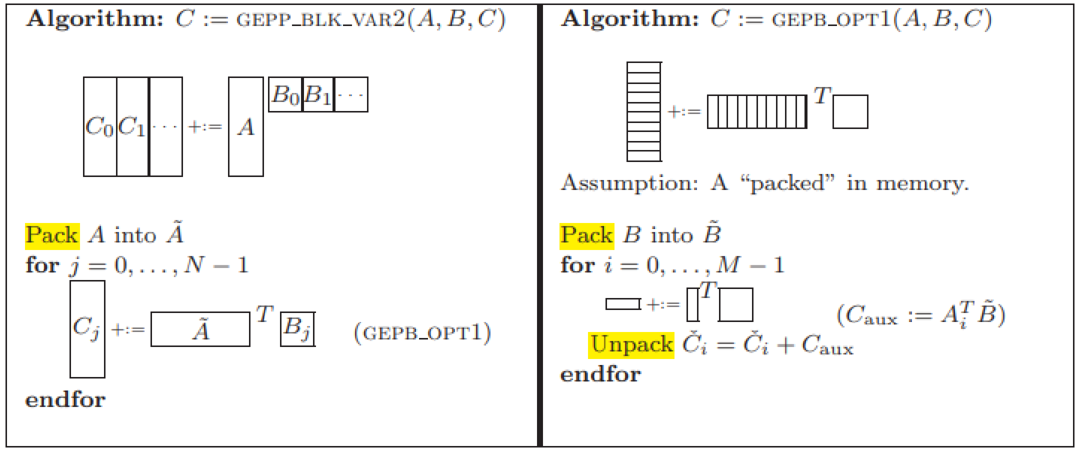
- Getting the formula right for blocking + repacking can be difficult. It’s worth spending some time to work it out. Even harder with multiple nested levels; may want to write a recursive “pack” procedure.

- Advanced: copy and pack “on-the-fly” into a memory region sized according to the cache, not the entire input matrix; produces a memory-efficient implementation if you choose the loop order to avoid $O(n^3)$ work.

- Packing so that elements accessed contiguously are contiguous in memory can also help you avoid translation lookaside buffer misses (page faults)
  - Processor can “hide” cache misses via instruction-level parallelism…
  - … but a TLB miss causes the entire pipeline to stall while the CPU fetches the correct page

Examples of Multilevel Blocking + Repacking

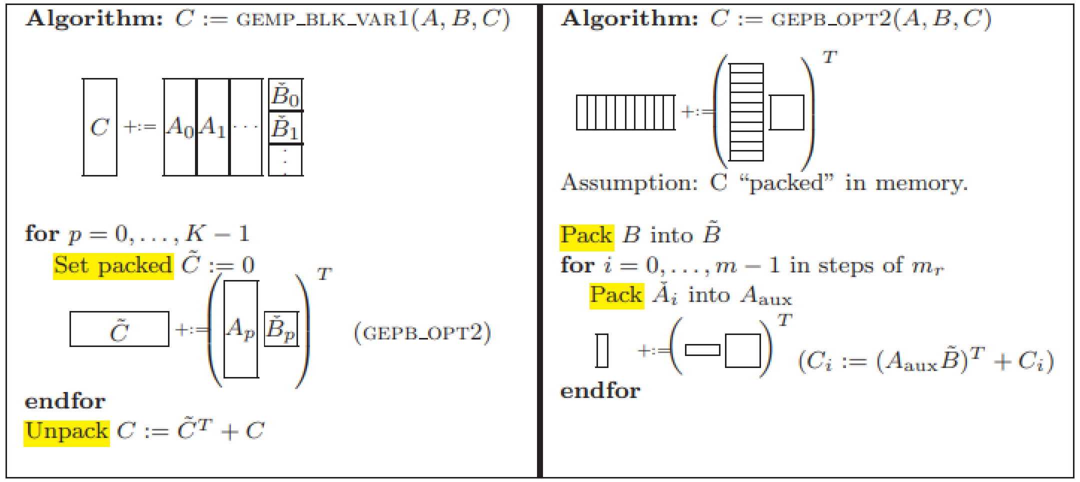
- From “Anatomy of High-Performance Matrix Multiplication”, linked on the assignment page
- Paper has more details on why each choice is desirable
- Several levels of blocking here: L2, L1, registers (micro-kernel)



```
Algorithm: $C := \text{GEPP\_BLK\_VAR2}(A, B, C)$

Pack $A$ into $\tilde{A}$
for $j = 0, \dots, N-1$
    $C_j \mathrel{+}:= \tilde{A}^T B_j$        (GEPB_OPT1)
endfor
```

```
Algorithm: $C := \text{GEPB\_OPT1}(A, B, C)$

Assumption: A “packed” in memory.

Pack $B$ into $\tilde{B}$
for $i = 0, \dots, M-1$
    ($C_{\text{aux}} := A_i^T \tilde{B}$)
    Unpack $\check{C}_i = \check{C}_i + C_{\text{aux}}$
endfor
```

Fig. 10. Optimized implementation of GEPP (left) via calls to GEPB_OPT1 (right).



```
Algorithm: $C := \text{GEMP\_BLK\_VAR1}(A, B, C)$

for $p = 0, \dots, K-1$
    Set packed $\tilde{C} := 0$
    $\tilde{C} \mathrel{+}:= (A_p \check{B}_p)^T$        (GEPB_OPT2)
endfor
Unpack $C := \tilde{C}^T + C$
```

```
Algorithm: $C := \text{GEPB\_OPT2}(A, B, C)$

Assumption: C “packed” in memory.

Pack $B$ into $\tilde{B}$
for $i = 0, \dots, m-1$ in steps of $m_r$
    Pack $\check{A}_i$ into $A_{\text{aux}}$
    ($C_i := (A_{\text{aux}} \tilde{B})^T + C_i$)
endfor
```

Pause

Questions so far? Tips get slightly more difficult from here.

Idea 4: Optimize your Loop Order

- **So far:** we've interpreted matrix multiplication as computing a "collection of dot products". Here is an alternate interpretation:

Matrix C is the sum of K outer products between *columns* of *A* and *rows* of *B*

- What does this mean in practice? For our naïve GEMM, just change the loop order

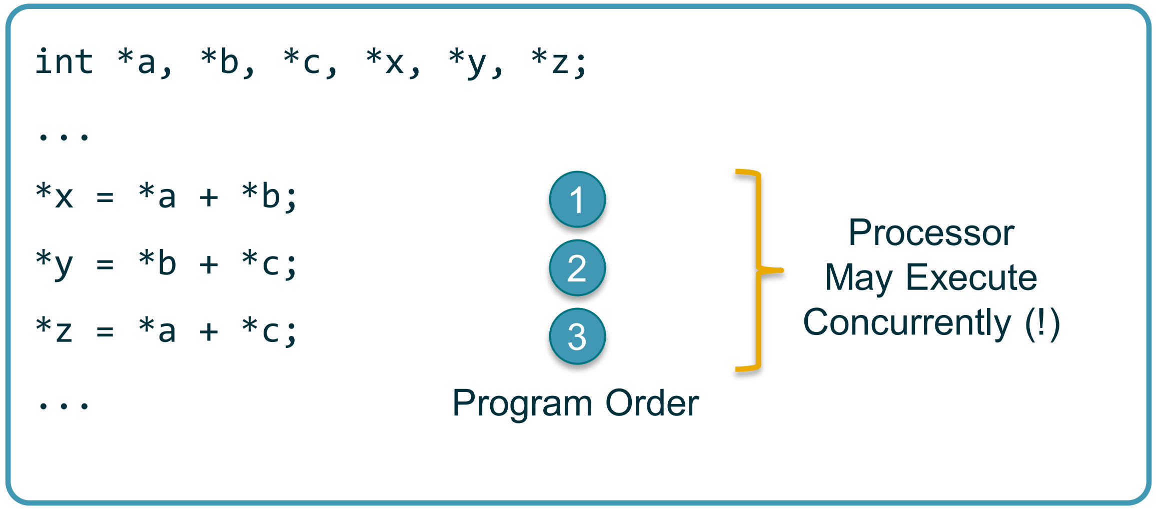
- Does this help? Somewhat… why are we doing this?

```
void simpleGEMM(...) {
 // Assume output matrix is 0
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      for(int k = 0; k < n; k++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

```
void simpleGEMMReordered(...) {
  // Assume output matrix is 0
  for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

Idea 4: Optimize your Loop Order

- **Answer:** We want to maximize **instruction-level parallelism** – especially if you write an optimized micro-kernel
- What is instruction-level parallelism? When we write code, we imagine statements execute one after the other
- The processor, on the other hand, is smart enough to realize that some instructions can be executed *out of order* without affecting correctness. It can schedule these operations to run concurrently

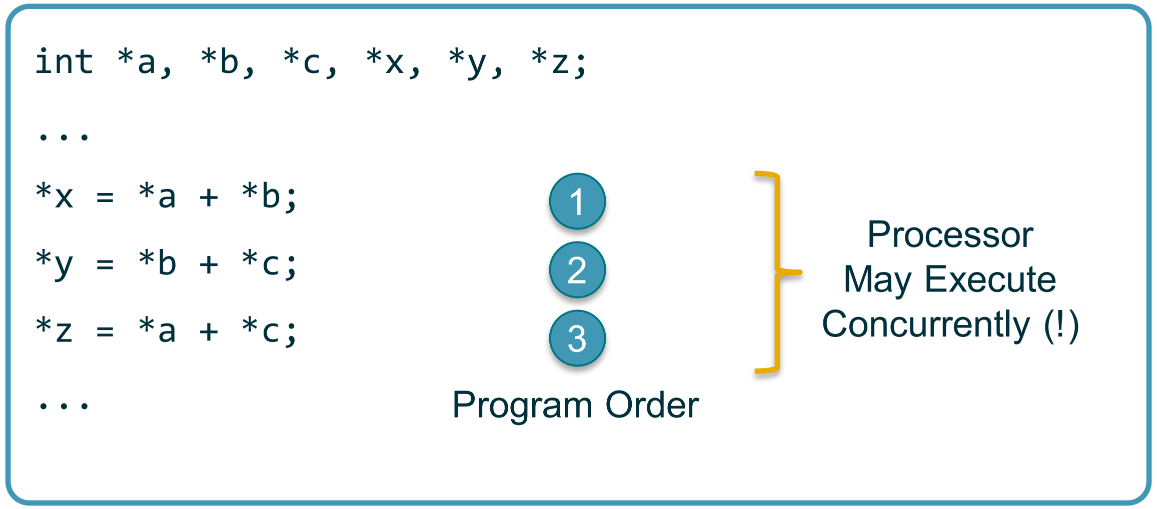


```
int *a, *b, *c, *x, *y, *z;
...
*x = *a + *b;
*y = *b + *c;
*z = *a + *c;
...
```

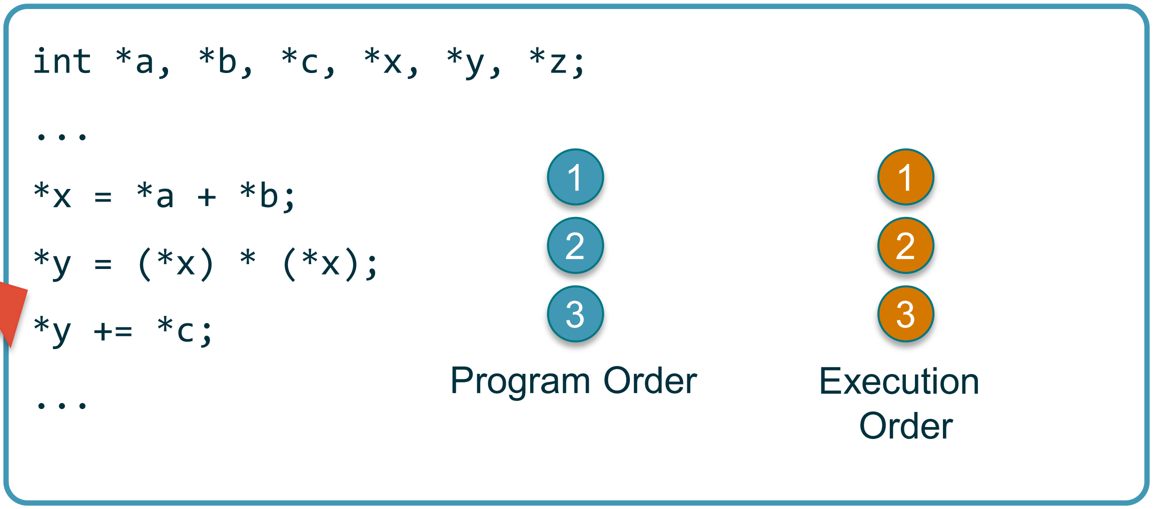
Idea 4: Optimize your Loop Order

- ILP can dramatically speed up performance
  - If an operation stalls on a cache miss, processor can continue execution until that memory load completes
  - KNL has two vector processing lanes; ILP keeps both busy.

- On the other hand, some programs impede ILP and out-of-order execution.



```
int *a, *b, *c, *x, *y, *z;
...
*x = *a + *b;
*y = *b + *c;
*z = *a + *c;
...
```



```
int *a, *b, *c, *x, *y, *z;
...
*x = *a + *b;
*y = (*x) * (*x);
*y += *c;
...
```

Dependencies force the processor to execute operations in sequence

Idea 4: Optimize your Loop Order

- Back to our example: in the original code, we cannot take advantage of ILP in the `k`-loop, since we are constantly overwriting `C[i + j * n]`.

- In the reordered code, each iteration of the inner two loops writes to a distinct location `C[i + j * n]`, so processor could potentially schedule them in parallel

- **BUT:** to really see the benefits of ILP, you should replace the inner loops with a sequence of explicit instructions. This is our next topic.

```
void simpleGEMM(...) {
 // Assume output matrix is 0
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      for(int k = 0; k < n; k++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

```
void simpleGEMMReordered(...) {
  // Assume output matrix is 0
  for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

Idea 5: Write a Micro-Kernel

- In the previous slide, we reordered our loops so that the processor could take advantage of instruction-level parallelism…

- But every multiply-and-add operation (also called fused multiply-add, or FMA) is followed by at least a loop counter increment: `j++`, in this case.

- This is an annoying. Processor also needs to keep checking whether the loop bounds have been exceeded, which is an expensive branch even if the branch predictor is good. Also impedes instruction-level parallelism (branch complicates instruction flow)

- **What is a micro-kernel for GEMM?** Performs matrix multiplication where at least one of the three input matrices is **very small** and fits into the processor registers. A micro-kernel usually contains long sequences of instructions.

Silly Micro-Kernel Example for Vector Addition

- To avoid giving away too much homework solution, lets look at a micro-kernel for another problem: **vector addition.** Given two *n*-length vectors stored in *A, *B, compute their elementwise sum and store it in *C. Here's the naïve code:

```
void addVecs(double *A, double *B, double *C, int n) {
    for(int i = 0; i < n; i++) {
        C[i] = A[i] + B[i];
    }
}
```

- Writing a micro-kernel will probably not speed up the naïve code for vector addition (why?), but it will give you an example you can heavily adapt for GEMM

- A good micro-kernel (with SIMD, which we will discuss shortly) can easily get you performance in the upper-end of the performance target window

Silly Micro-Kernel Example for Vector Addition

- Our toy micro-kernel will add together vectors of length 8.
- For those of you familiar with #pragma unroll, this is like manual loop unrolling
- Assume for now that *n* is divisible by 8 (if not, tack on an extra loop to the end to handle the tail, or pad the input with zeros)
- Let's rewrite the code so that we can call our micro-kernel easily:

```
void addVecs(double *A, double *B, double *C, int n) {
    for(int i = 0; i < n; i += 8) {
        double* aLoc = &(A[i]);
        double* bLoc = &(B[i]);
        double* cLoc = &(C[i]);
        for(int j = 0; j < 8; j++) {
            cLoc[j] = aLoc[j] + bLoc[j];
        }
    }
}
```

Our micro-kernel will do this!

Silly Micro-Kernel Example for Vector Addition

- Our toy micro-kernel will add together vectors of length 8.
- For those of you familiar with #pragma unroll, this is like manual loop unrolling
- Assume for now that *n* is divisible by 8 (if not, tack on an extra loop to the end to handle the tail, or pad the input with zeros)
- Let’s rewrite the code so that we can call our micro-kernel easily:

```
void addVecs(double *A, double *B, double *C, int n) {
    for(int i = 0; i < n; i += 8) {
        double* aLoc = &(A[i]);
        double* bLoc = &(B[i]);
        double* cLoc = &(C[i]);
        micro_kernel(aLoc, bLoc, cLoc);
    }
}
```

Silly Micro-Kernel Example for Vector Addition

- Microkernel flow:
  - **Declare** register variables
  - **Load** inputs from memory
  - **Compute**
  - **Store** output registers to memory

- The code to the right **ignores** the fact that we only have 16 registers in x86-64 available to us, so not the most efficient
  - Not all local variables will persist in registers, compiler will optimize
  - SIMD (coming up) will fix this

```
void micro_kernel (double* A, double* B, double* C) {
     // Declare
     double A0, A1, A2, A3, A4, A5, A6, A7;
     double B0, B1, B2, B3, B4, B5, B6, B7;
     double C0, C1, C2, C3, C4, C5, C6, C7;

     // Load
     A0 = A[0]; ... ; A7 = A[7];
     B0 = B[0]; ... ; B7 = B[7];

     // Compute
     C0 = A0 + B0; ...; C7 = A7 + B7;

     // Store
     C[0] = C0; ... ; C[7] = C7;
}
```

Silly Micro-Kernel Example for Vector Addition

- What does this buy us?
  - Only need to increment loop counter every 8 iterations
  - Lots of arithmetic operations right next to each other, good ILP
- *This* example is terrible, though… no need to load A and B to registers with explicit code, since they are only used once, and the compiler does it anyway
- But for GEMM, we reuse A, B, C **several** times in the compute step. Explicit loads / stores promote register reuse

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    double A0, A1, A2, A3, A4, A5, A6, A7;
    double B0, B1, B2, B3, B4, B5, B6, B7;
    double C0, C1, C2, C3, C4, C5, C6, C7;

    // Load
    A0 = A[0]; ... ; A7 = A[7];
    B0 = B[0]; ... ; B7 = B[7];

    // Compute
    C0 = A0 + B0; ...; C7 = A7 + B7;

    // Store
    C[0] = C0; ... ; C[7] = C7;
}
```

This sequence is WAY longer for GEMM!

Micro-Kernel Tips

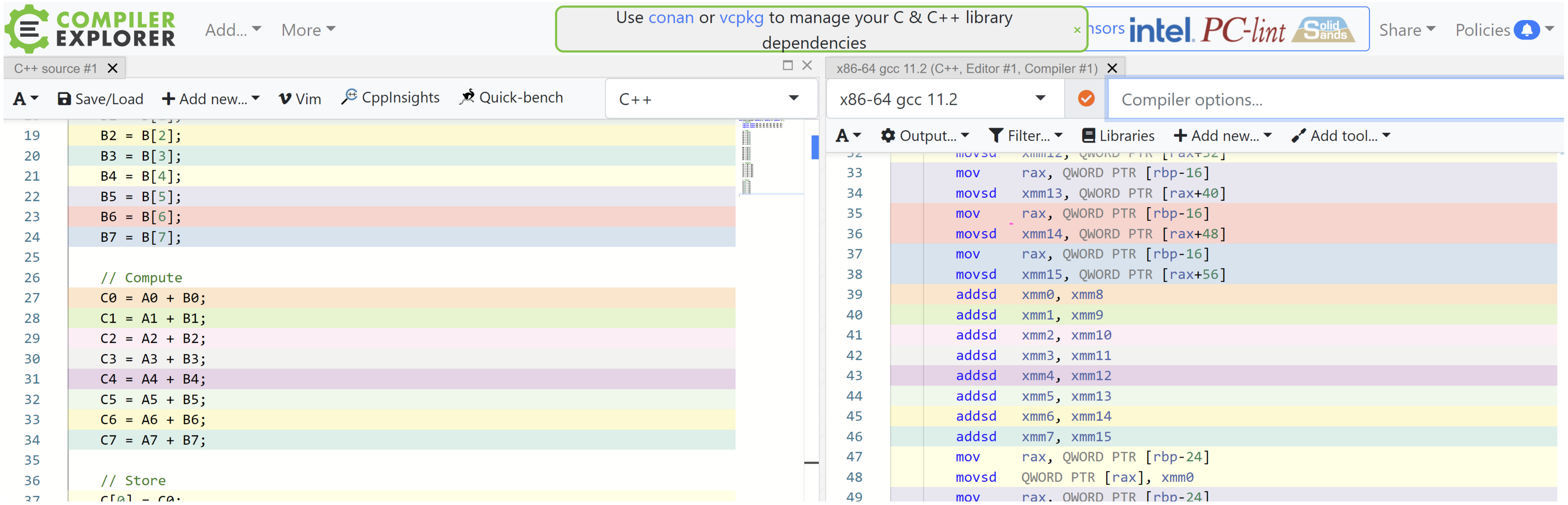
- You're going to write a lot of repetitive code. Use Python or tool of your choice to **auto-generate** C code for several matrix sizes (e.g. 2 x 2, 4 x 4, 8 x 8 GEMMs)

- Personal favorite: Cog (https://nedbatchelder.com/code/cog. Extremely simple. Uses Python, can install on Cori with `pip install cogapp --user`). You can do things like this inline in a source file, call `cog –r <myfile>` to generate the output. Read the documentation for more.

```
// This is my C++ file.
...
/*[[[cog
import cog
fnames = ['DoSomething', 'DoAnotherThing', 'DoLastThing']
for fn in fnames:
    cog.outl("void %s();" % fn)
]]]*/
void DoSomething();
void DoAnotherThing();
void DoLastThing();
//[[[end]]]
...
```

Micro-Kernel Tips

- You should use the Compiler Explorer (https://godbolt.org/) to check out the Assembly that your compiled function is producing. Here is what it does for our vector addition micro-kernel:



```
    B2 = B[2];
    B3 = B[3];
    B4 = B[4];
    B5 = B[5];
    B6 = B[6];
    B7 = B[7];

    // Compute
    C0 = A0 + B0;
    C1 = A1 + B1;
    C2 = A2 + B2;
    C3 = A3 + B3;
    C4 = A4 + B4;
    C5 = A5 + B5;
    C6 = A6 + B6;
    C7 = A7 + B7;

    // Store
```

```
        mov     rax, QWORD PTR [rbp-16]
        movsd   xmm13, QWORD PTR [rax+40]
        mov     rax, QWORD PTR [rbp-16]
        movsd   xmm14, QWORD PTR [rax+48]
        mov     rax, QWORD PTR [rbp-16]
        movsd   xmm15, QWORD PTR [rax+56]
        addsd   xmm0, xmm8
        addsd   xmm1, xmm9
        addsd   xmm2, xmm10
        addsd   xmm3, xmm11
        addsd   xmm4, xmm12
        addsd   xmm5, xmm13
        addsd   xmm6, xmm14
        addsd   xmm7, xmm15
        mov     rax, QWORD PTR [rbp-24]
        movsd   QWORD PTR [rax], xmm0
```

Pause

Questions? We’ll be covering Single-Instruction-Multiple Data instructions next, which are potentially very useful to speed up performance.

Idea 6: SIMD Micro-Kernel

- **SIMD: S**ingle **I**nstruction **M**ultiple **D**ata. Issue one instruction to perform several arithmetic operations. Available to you: 512-bit Intel Advanced Vector eXtensions (AVX512) **intrinsics**
- In theory, the compiler can look at loops and generate vectorized code itself. But GEMM is complicated enough that you'll probably have to do it yourself. Serious performance boost if you do it right

A Crash Course in AVX512

- AVX512 intrinsics are “functions” that can operate on **vector registers**. Each KNL core has 32 vector registers at its disposal.

- Intrinsics don’t have overhead of actual function calls. They are just nice wrappers for low-level Assembly commands (i.e. 1 intrinsic line = 1 assembly instruction)

- Each register is 512 bits wide, fits 8 double-precision 64-bit floats.

- The HW1 CMakeLists.txt is set up so that most (but not all) AVX512 intrinsics become available to you by writing

```
#include <immintrin.h>
```

Silly Micro-Kernel Revisited with AVX512

- Let’s rewrite our toy micro-kernel using AVX512. Don’t expect a speedup for vector addition, though (we are bandwidth-bound).

- **First step**: let’s replace our local variables with vectors.

- Vector datatypes are prefixed with __m512 and followed by either:
  - <nothing>: If the vector contains 16 single precision floating point numbers
  - d: If the vector contains 8 double precision floating point numbers
  - i: If the vector contains integers (any precision)

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    double A0, A1, A2, A3, A4, A5, A6, A7;
    double B0, B1, B2, B3, B4, B5, B6, B7;
    double C0, C1, C2, C3, C4, C5, C6, C7;

    // Load
    A0 = A[0]; ... ; A7 = A[7];
    B0 = B[0]; ... ; B7 = B[7];

    // Compute
    C0 = A0 + B0; ...; C7 = A7 + B7;

    // Store
    C[0] = C0; ... ; C[7] = C7;
}
```

Silly Micro-Kernel Revisited with AVX512

- Let's rewrite our toy micro-kernel using AVX512. Don't expect a speedup for vector addition, though (we are bandwidth-bound).

- **First step**: let's replace our local variables with vectors.

- Vector datatypes are prefixed with __m512 and followed by either:
  - <nothing>: If the vector contains 16 single precision floating point numbers
  - d: If the vector contains 8 double precision floating point numbers
  - i: If the vector contains integers (any precision)

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    __m512d Ar;
    __m512d Br;
    __m512d Cr;

    // Load
    A0 = A[0]; ... ; A7 = A[7];
    B0 = B[0]; ... ; B7 = B[7];

    // Compute
    C0 = A0 + B0; ...; C7 = A7 + B7;

    // Store
    C[0] = C0; ... ; C[7] = C7;
}
```

Silly Micro-Kernel Revisited with AVX512

- **Second step:** load input registers with data.
- We will use the double-precision load intrinsic:

```
__m512d _mm512_load_pd (void const* mem_addr)
```

  - **Input:** Pointer to an **aligned,** contiguous segment of 8 doubles in memory
  - **Returns:** A vector of 8 doubles
- Again: not a real function call, just looks like one

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    __m512d Ar;
    __m512d Br;
    __m512d Cr;

    // Load
    Ar = _mm512_load_pd(A);
    Br = _mm512_load_pd(B);

    // Compute
    C0 = A0 + B0; ...; C7 = A7 + B7;

    // Store
    C[0] = C0; ... ; C[7] = C7;
}
```

How to Read an Intrinsic Signature

```
__m512d _mm512_load_pd(void const *addr)
```

This intrinsic returns a vector of 8 doubles

This is an AVX512 intrinsic (not AVX2)

I want to **load** some **aligned** data from memory into a vector (use unaligned version otherwise)

My data is stored in double precision

Here’s a pointer to the first of 8 data words in memory

- Signature looks like garbage at first, but each part has meaning

Silly Micro-Kernel Revisited with AVX512

- **Third step:** perform computation

- The double precision vector add instruction:

```
__m512d _mm512_add_pd (__m512d a, __m512d b)
```

  - **Inputs:** Two vectors with 8 doubles each
  - **Returns:** A vector of 8 doubles with elementwise sums

```
void micro_kernel (double* A, double* B, double* C) {
     // Declare
     __m512d Ar;
     __m512d Br;
     __m512d Cr;

     // Load
     Ar = _mm512_load_pd(A);
     Br = _mm512_load_pd(B);

     // Compute
     Cr = _mm512_add_pd(Ar, Br);

     // Store
     C[0] = C0; ... ; C[7] = C7;
}
```

Silly Micro-Kernel Revisited with AVX512

- **Fourth step:** store back the output

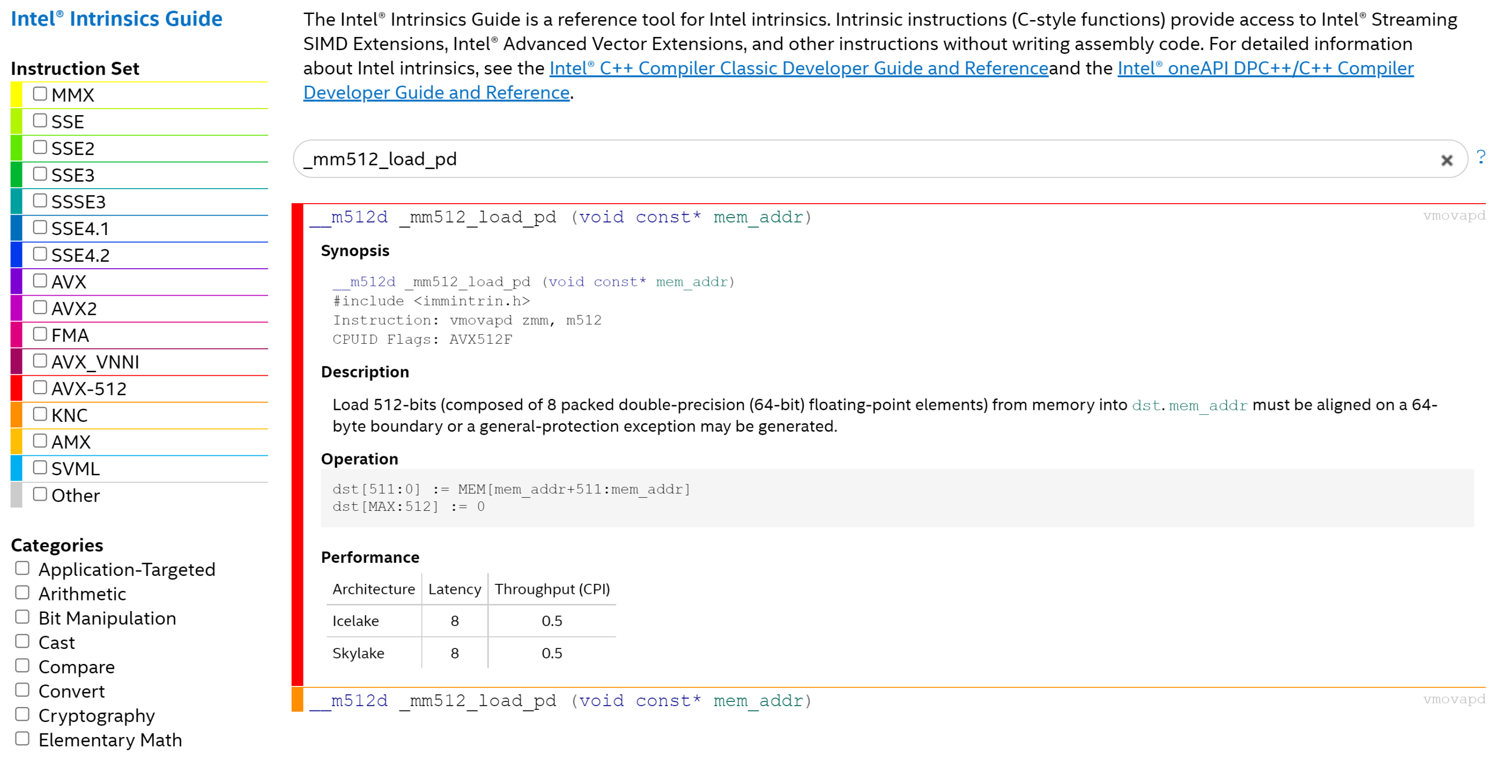
- The double precision vector store instruction:

```
void _mm512_store_pd (void* addr, __m512d a)
```

  - **Inputs:** Pointer to **aligned** memory location and a vector of 8 doubles
  - **Postcondition:** Store operation executed

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    __m512d Ar;
    __m512d Br;
    __m512d Cr;

    // Load
    Ar = _mm512_load_pd(A);
    Br = _mm512_load_pd(B);

    // Compute
    Cr = _mm512_add_pd(Ar, Br);

    // Store
    _mm512_store_pd(C, Cr);
}
```

Where to Look Up Intrinsic Signatures



Intel® Intrinsics Guide

Instruction Set

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math

The Intel® Intrinsics Guide is a reference tool for Intel intrinsics. Intrinsic instructions (C-style functions) provide access to Intel® Streaming SIMD Extensions, Intel® Advanced Vector Extensions, and other instructions without writing assembly code. For detailed information about Intel intrinsics, see the Intel® C++ Compiler Classic Developer Guide and Referenceand the Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference.

_mm512_load_pd ✕ ?

```
__m512d _mm512_load_pd (void const* mem_addr)                    vmovapd
```

Synopsis

```
__m512d _mm512_load_pd (void const* mem_addr)
#include <immintrin.h>
Instruction: vmovapd zmm, m512
CPUID Flags: AVX512F
```

Description

Load 512-bits (composed of 8 packed double-precision (64-bit) floating-point elements) from memory into `dst`. `mem_addr` must be aligned on a 64-byte boundary or a general-protection exception may be generated.

Operation

```
dst[511:0] := MEM[mem_addr+511:mem_addr]
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	8	0.5
Skylake	8	0.5

```
__m512d _mm512_load_pd (void const* mem_addr)                    vmovapd
```

- Intel Intrinsics Explorer (https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html). Let's you look up intrinsics, filter by type, etc. Super useful for this assignment.

SIMD Micro-Kernel Tips for GEMM

- Worth your time. You'll need the **set1_pd, fmadd_pd** intrinsics, at least, in addition to the ones we've already covered. Try different micro-kernel shapes (8 x 8, 16 x 16, etc.)

- Remember the reordered GEMM code? Imagine what would happen if you wrote out all the statements in the inner `i, j` loops one after the other for small (e.g. 8 x 8, 16 x 16) matrices. Can you concisely express that sequence of ops with AVX512? Python code generator still helpful.

```
void simpleGEMMReordered(...) {
  // Assume output matrix is 0
  for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

Pause

Phew – almost there. Don’t worry if you can’t get to these last optimizations – although they can yield a significant speedup if you do them right.

Idea 7: Software Prefetching

- Prefetching brings memory into the cache before an instruction explicitly requests it. A few judicious software prefetch commands boosts GEMM by 5-10%. Pretty good on KNL!

- Check out the `_mm_prefetch` intrinsic:
  - Fetches a line at the given memory location into the cache level suggested by the hint
  - Fun fact: Don't have to worry about providing bad memory locations! Illegal / out-of-bounds prefetches are silently ignored (you are suggesting to the processor, not commanding it).
- **Warnings:**
  - Prefetching has little to no benefit if you are compute-bound or if your blocking is not optimal. Treat as icing on the cake.
  - Too many prefetch instructions will slow your code down
  - Useless if you don't search for the optimal prefetch distance (see the next slide)

Another Silly Example: Prefetching for Vector Addition

- Probably won't result in a speedup for vector addition – but a similar approach WILL yield a speedup for GEMM.

- At each iteration, we prefetch a line of 8 elements containing location `A[i + 48]` into the L1 cache (that's what the locality hint _MM_HINT… means. Check out this page for all hints). Similar for B.

- Don't need to write an optimized micro-kernel to try prefetching, but more likely to see a benefit if you have

```
#define PFETCH_DIST 48
void addVecs(double *A, double *B, double *C, int n) {
     for(int i = 0; i < n; i += 8) {
          double* aLoc = &(A[i]);
          double* bLoc = &(B[i]);
          double* cLoc = &(C[i]);
          _mm_prefetch(aLoc + PFETCH_DIST, _MM_HINT_T0);
          _mm_prefetch(bLoc + PFETCH_DIST, _MM_HINT_T0);
          micro_kernel(aLoc, bLoc, cLoc);
     }
}
```

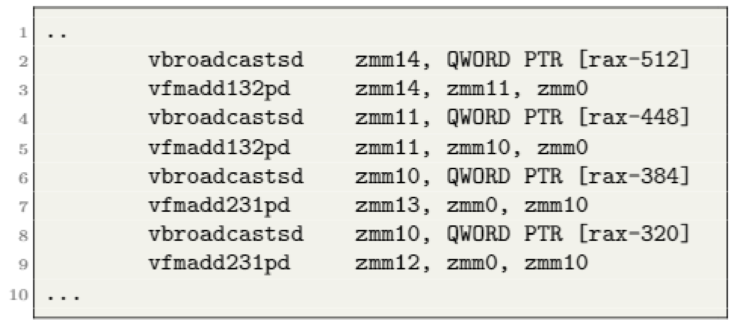
Tuning the Prefetch Distance

- Key parameter is **prefetch distance**: how far in advance do you want to bring in the cache line?
- **Prefetch distance too small:** Data won't be available by the time you need it
- **Prefetch distance too large:** Data might be evicted from cache by the time you need it, clogs up the instruction stream
- There is a sweet spot where you get optimal performance, but you'll need to perform a line search to find it.
  - Exhaustive search probably better than binary search, which could get stuck in local minima

Idea 8: Write Inline Assembly

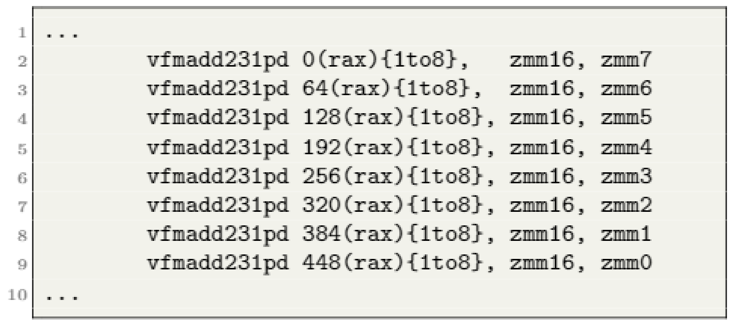
- If you've gotten this far, you may have realized that matrix multiplication depends on two AVX512 intrinsics:
  - `_mm512_fmadd_pd`
  - `_mm512_set1_pd.` This is called a **broadcast intrinsic**

- GCC 8.3 has a major flaw: it does not support "embedded broadcast", which fuses these operations in assembly. This causes a computation bottleneck: best you can hope for is 22.4 GFLOPs, even if you do everything else right!



```
..
        vbroadcastsd    zmm14, QWORD PTR [rax-512]
        vfmadd132pd     zmm14, zmm11, zmm0
        vbroadcastsd    zmm11, QWORD PTR [rax-448]
        vfmadd132pd     zmm11, zmm10, zmm0
        vbroadcastsd    zmm10, QWORD PTR [rax-384]
        vfmadd231pd     zmm13, zmm0, zmm10
        vbroadcastsd    zmm10, QWORD PTR [rax-320]
        vfmadd231pd     zmm12, zmm0, zmm10
...
```

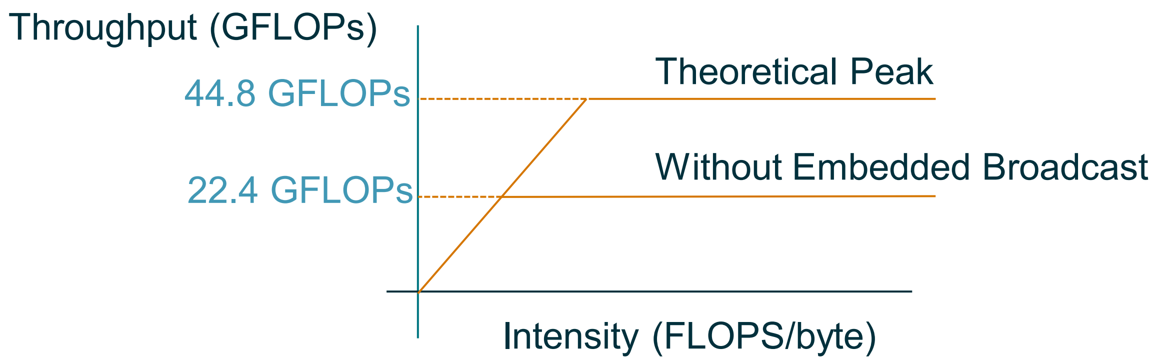
Listing 1: GCC 8.3 ASM for FMA Intrinsics



```
...
        vfmadd231pd 0(rax){1to8},   zmm16, zmm7
        vfmadd231pd 64(rax){1to8},  zmm16, zmm6
        vfmadd231pd 128(rax){1to8}, zmm16, zmm5
        vfmadd231pd 192(rax){1to8}, zmm16, zmm4
        vfmadd231pd 256(rax){1to8}, zmm16, zmm3
        vfmadd231pd 320(rax){1to8}, zmm16, zmm2
        vfmadd231pd 384(rax){1to8}, zmm16, zmm1
        vfmadd231pd 448(rax){1to8}, zmm16, zmm0
...
```

Listing 2: Our ASM with Embedded Broadcasts

Idea 8: Write Inline Assembly



- The only way to fix this: write the correct inline assembly yourself (i.e. replace the intrinsics)

- WARNING: You need to make sure your implementation is correct when it compiles on our end! You're on your own here. To get started, read:
  - https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

- DO NOT do this until you have written the code with AVX512 intrinsics! Inline assembly will be a drop-in substitute. Write intrinsics, then replace with inline ASM as icing on the cake.

Silly Micro-Kernel Revisited with Inline Assembly

- Let's rewrite the silly micro-kernel one last time, but we're going to replace the intrinsic with inline ASM
- We will NOT get a speedup here; intrinsics are just as good as inline ASM in almost every case, and usually even better.
- The embedded broadcast bug that we're trying to solve has been fixed in later versions of GCC.
- Writing inline ASM is a useful skill to have in your kit, though… so if you want a challenge, here's how.

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    __m512d Ar;
    __m512d Br;
    __m512d Cr;

    // Load
    Ar = _mm512_load_pd(A);
    Br = _mm512_load_pd(B);

    // Compute
    Cr = _mm512_add_pd(Ar, Br);

    // Store
    _mm512_store_pd(C, Cr);
}
```

Silly Micro-Kernel Revisited with Inline Assembly

- **Aligned** load and store operations done by `vmovapd`
    - Load: provide memory location, then register
    - Store: provide register, then memory location

- Addition performed by `vaddpd` ; supply two inputs followed by output
- First colon, then a list of symbol definitions and constraints for output / input registers
    – "+v" denotes a read / write AVX512 vector register
- Second colon, then a list of input registers
    – "r" denotes a RO standard x86-64 register
- Third colon: memory clobber (important!)
    – Tells the compiler we're reading / writing memory, make sure no values cached in registers before

```
void micro_kernel (double *A, double *B, double *C) {
    __m512d Ar, Br, Cr;
    asm volatile (
        "vmovapd (%[A]), %[Ar]\n\t"        // Load A
        "vmovapd (%[B]), %[Br]\n\t"        // Load B
        "vaddpd %[Ar], %[Br], %[Cr]\n\t"   // Add
        "vmovapd %[Cr], (%[C])\n\t"        // Store C
        : [Cr] "+v" (Cr),
          [Ar] "+v" (Ar),
          [Br] "+v" (Br)
        : [A] "r" (A),
          [B] "r" (B),
          [C] "r" (C)
        : "memory"
    );
}
```

Some Tips for Inline ASM

- Test your code thoroughly. For example, not including the memory clobber causes some ASM code to work correctly at O1, but not at O3.
- Try to include as many ASM statements in one block as possible. Entering an ASM block is expensive (compiler needs to commit any registers that cache memory, etc.)
- **Code generation** is the way to go here. Don’t bother typing everything out.

GEMM in Practice (Stolen from Jim)

- High performance matrix multiplication code uses all the techniques discussed

- A short history of the technique:
  - PHiPAC (Portable High Performance ANSI C) was one of the earliest projects in this area (done at Berkeley), for tuning matrix multiplication. 1997 conference paper recently won a Test-of-Time award.
  - ATLAS (Automatically Tuned Linear Algebra Software) is another project that started about the same time, aimed at tuning all the BLAS, and is ongoing, and also won a Test-of-Time award.
  - BLIS and OpenBLAS are two more recent and ongoing projects.
  - OSKI (Optimized Sparse Kernel Interface) autotunes SpMV (sparse matrix times dense vector multiplication), in part by choosing an optimal sparse data structure for each sparse matrix. POSKI is a parallel version.

Good luck! Break the record!

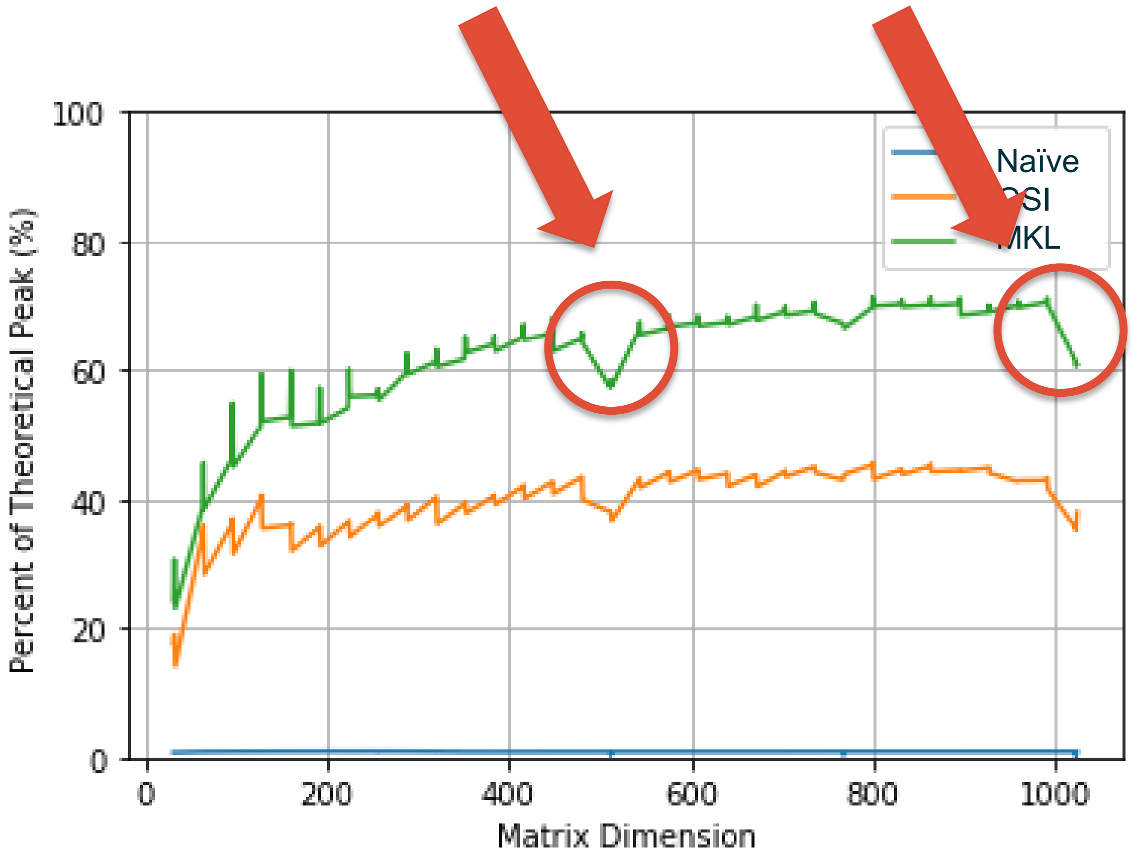
There are some extra slides after this one. You might want to look at them.

Extra Idea: Dealing with Powers of Two

- This one is simpler than the ones that came before it. Let's go back to that performance graph at the beginning of the presentation. Any ideas what's going on here?

- At least one good way to fix this mentioned earlier in the presentation

- Performance dip unavoidable, but you can mitigate it to boost average performance by 1-2% on the homework benchmark.



More on Why Loop Order Optimization is Good

- **Recall:** we have 32 vector registers available to us. For now, let's **assume** that matrix dimensions are so small that all of one matrix + 2 rows / columns will fit into registers

- Let's think about how a (very, very intelligent) compiler would generate instructions for the **original** GEMM code:

```
void simpleGEMM(...) {
 // Assume output matrix is 0
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      for(int k = 0; k < n; k++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

Compiler

```
simpleGemm:
  BREGS <- LOAD ALL OF B FROM MEMORY
  LOOP i:
    CREG <- LOAD ROW i OF C FROM MEMORY
    AREG <- LOAD ROW i OF A FROM MEMORY
    LOOP j:
      LOOP k:
        FMADD INSTRUCTION
     CREG -> STORE ROW i of C TO MEMORY
```

STORE inside i loop

More on Why Loop Order Optimization is Good

- How about the **reordered** GEMM code?
- We just got a STORE operation out of the loop – good news!

```
void simpleGEMMReordered(...) {
  // Assume output matrix is 0
  for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         C[i + j * n] += A[i + k * n] * B[k + j * n];
...
```

Compiler

Outside the loop!

```
simpleGemm:
  CREGS <- LOAD ALL OF C FROM MEMORY
  LOOP k:
    AREG <- LOAD COL k OF A FROM MEMORY
    AREG <- LOAD ROW k OF B FROM MEMORY
    LOOP i:
      LOOP j:
        FMADD INSTRUCTION
   CREGS -> STORE ALL OF C TO MEMORY
```