The joys of Matrix Multiplication

Will
Ocean Park
101
Long Island
4
Willapa National Wildlife Refuge...
Naselle
Ro
Ilwaco
101
401
Astoria
Sven
101
Lewis and Clark National Historical Park

Looking Southeast from Here

Reasoning About Performance: Big O notation

- When we talk about performance we typically simplify the discussion by only considering how the performance varies with problem-size for the case of a large problem.

- In other words, we consider the asymptotic performance for large problems and ignore smaller performance effects that don’t scale with the problem size.

- Consider Matrix Multiplication:

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * B[k*N+j];
       }
     }
   }
}
```

Cost is determined by FLOPs and memory movement:

$2*n^3 = O(n^3)$ FLOPS

Operates on $3*n^2 = O(n^2)$ numbers

Reasoning About Performance: Big O notation

- When we talk about performance we typically simplify the discussion by only considering how the performance varies with problem-size for the case of a large problem.
- In other words, we consider the asymptotic performance for large problems and ignore smaller performance effects that don't scale with the problem size.
- Consider Matrix Multiplication:

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * B[k*N+j];
       }
     }
   }
}
```

Cost is determined by FLOPs and memory movement:

$2*n^3 = O(n^3)$ FLOPS

Operates on $3*n^2 = O(n^2)$ numbers

We love problems where the compute scales at a higher power than memory movement … that means if there is enough memory in the system, we can grow the problem to a large enough size to make the problem compute-bound.

Multiplication of dense matrices can achieve 95+ percent of peak performance!

Matrix Multiplication

- I am impressed at the creative ways you came up with to solve the matrix multiplication problem.

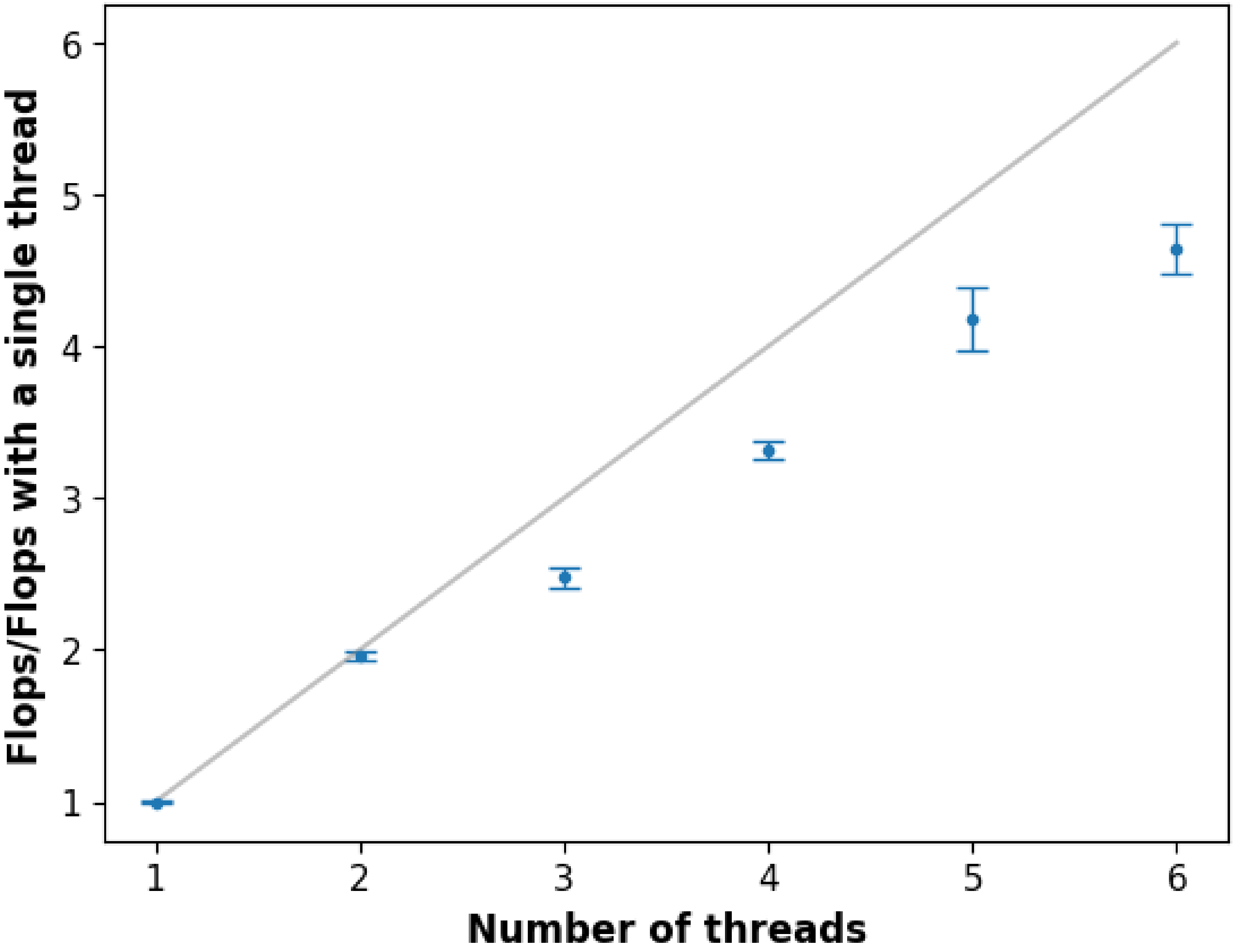
- To have a common point of comparison I ran everything on my Apple M2 laptop
  - 4 Performance Cores
  - 4 Efficiency Cores
  - 102.4 Gbytes/sec memory bandwidth

Matrix Multiplication Speedup Plot

One person provided a speedup plot.

This is great.

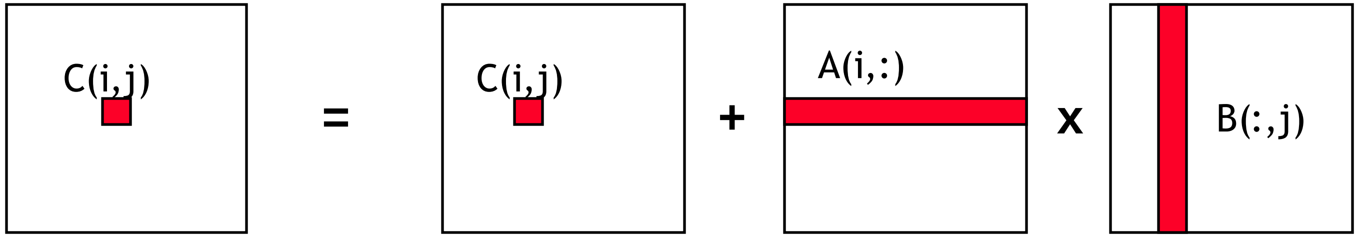
You MUST always check where your solution speedup falls off



Matrix Multiplication

- The common way we write matrix multiply (the ijk algorithm … named for the standard order of loops) results in a pattern of dot-products over rows and columns

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * B[k*N+j];
       }
     }
   }
}
```



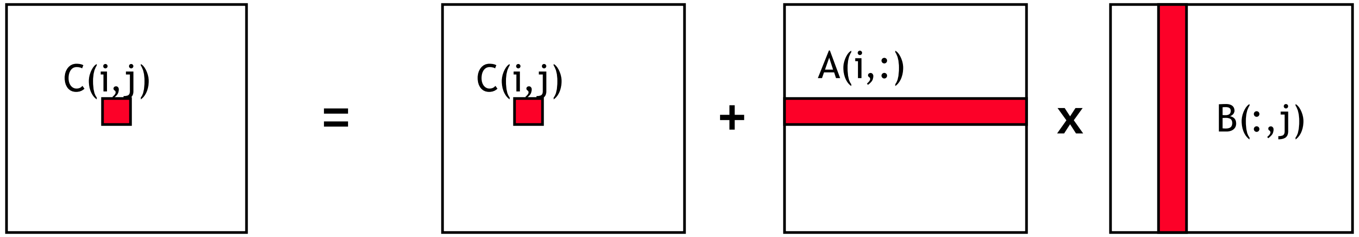
Dot product of a row of A and a column of B for each element of C

Matrix Multiplication

- The common way we write matrix multiply (the ijk algorithm … named for the standard order of loops) results in a pattern of dot-products over rows and columns

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   #pragma omp parallel for
   for (int i = 0; i < N; i++) {
     for (int j = 0; j < N; j++) {
       for (int k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * B[k*N+j];
       }
     }
   }
}
```

3253 MFLOPS with 8 threads and matrices of size 300, 600, 900



Dot product of a row of A and a column of B for each element of C

The worst performance

- Serial loop over i, parallelize the second loop

```
void mm_opt(int Ndim, int Mdim, int Pdim, TYPE *A, TYPE *B, TYPE *C) {
  int i;

  for (i = 0; i < Ndim; i++) {
   #pragma omp parallel for   // schedule(static,30)
    for (int j = 0; j < Mdim; j++) {
      TYPE tmp = 0.0;
      for (int k = 0; k < Pdim; k++) {
        /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
        tmp += *(A + (i * Pdim + k)) * *(B + (k * Mdim + j));
      }
      *(C + (i * Mdim + j)) += tmp;
    }
  }
}
```

2377 MFLOPS with 8 threads and matrices of size 300, 600, 900

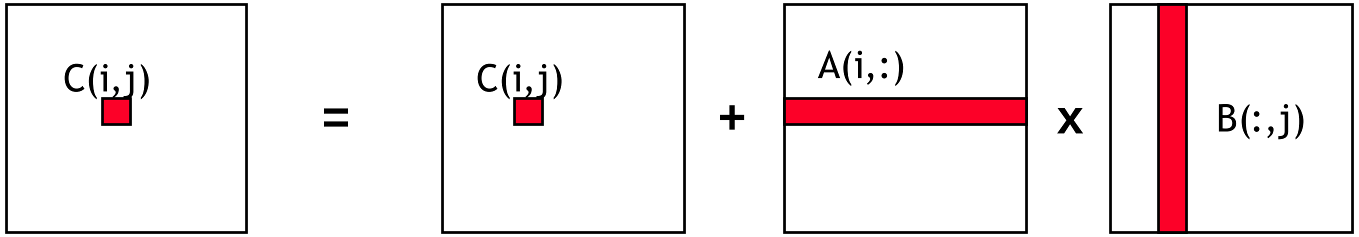
This creates extreme levels of thread management overhead since you Fork and Join the team of thread Ndim times.

Matrix Multiplication

- The common way we write matrix multiply (the ijk algorithm … named for the standard order of loops) results in a pattern of dot-products over rows and columns

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * B[k*N+j];
       }
     }
   }
}
```

For row based matrix storage (the common approach in C) access to A is cache aligned but B uses one value from each cache lines and marches across large strides in memory (hence causing TLB misses).



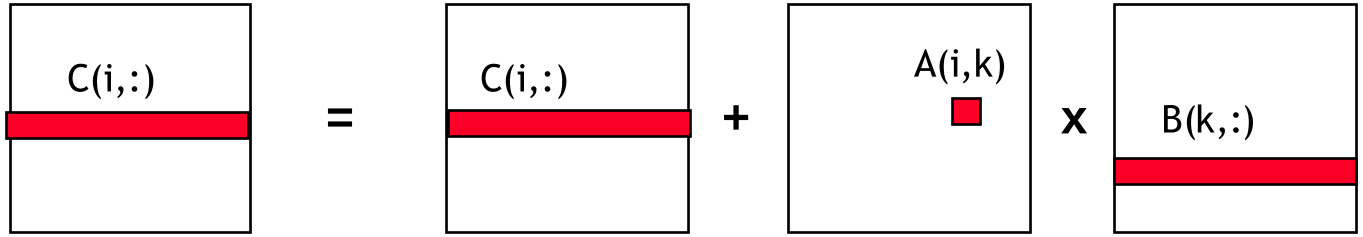
Dot product of a row of A and a column of B for each element of C

Matrix Multiplication: ikj algorithm

- We can reduce cache misses by swapping the j and k loops … so the innermost loop is over j and the access patterns across the C and B matrices cache friendly. We call this is ikj algorithm

```
void mat_mul(int N, float *A, float *B, float *C)
{
  int i, j, k;
  for (i = 0; i < N; i++) {
   for (k = 0; k < N; k++) {
     for (j = 0; j < N; j++) {
       C[i*N+j] += A[i*N+k] * B[k*N+j];
     }
   }
  }
}
```

Update a full row of C by scaling a row of B with an element from A



Dot product of a row of A and a column of B for each element of C

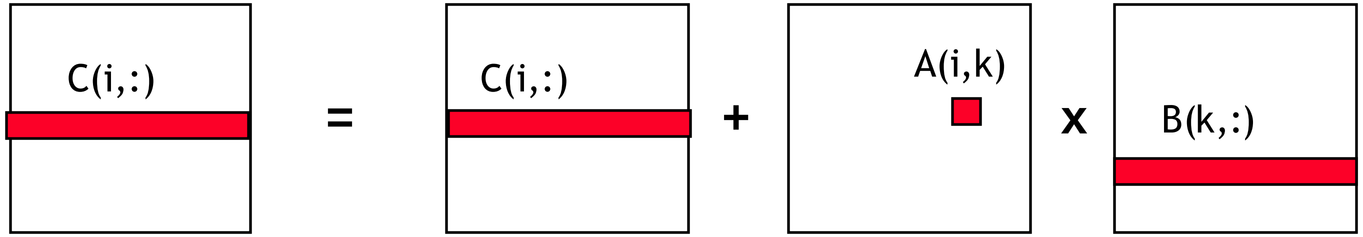
Matrix Multiplication: ikj algorithm

- We can reduce cache misses by swapping the j and k loops … so the innermost loop is over j and the access patterns across the C and B matrices cache friendly. We call this is ikj algorithm

```
void mat_mul(int N, float *A, float *B, float *C)
{
   int i, j, k;
   #pragma omp parallel for
   for (i = 0; i < N; i++) {
     for (k = 0; k < N; k++) {
         for (j = 0; j < N; j++) {
             C[i*N+j] += A[i*N+k] * B[k*N+j];
         }
      }
   }
}
```

Update a full row of C by scaling a row of B with an element from A

3..3 GFLOPS for matrices of size 300, 600, 900



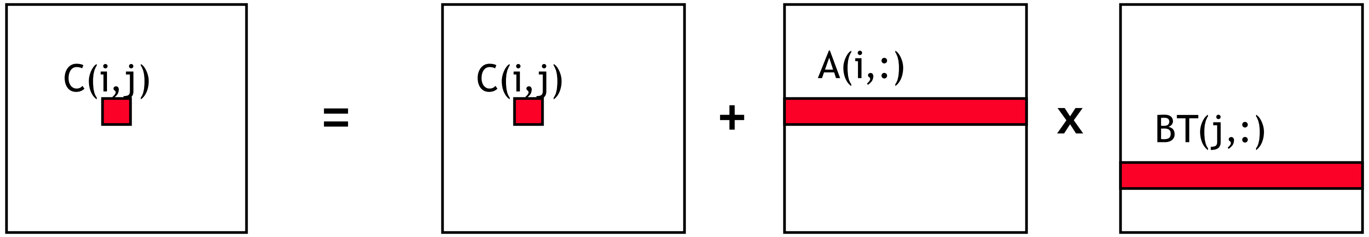
Dot product of a row of A and a column of B for each element of C

Matrix Multiplication: Transpose B first

- The common way we write matrix multiply (the ijk algorithm … named for the standard order of loops)

```
void mat_mul(int N, float *A, float *BT, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k++) {
         C[i*N+j] += A[i*N+k] * BT[j*N+k];
       }
     }
   }
}
```

Both involve Cache Friendly stride one access patterns



Dot product of a row of A and a column of B for each element of C

Matrix Multiplication: Transpose B first

- The consistently best approach surprised me … transpose B and unroll the innermost loop to help the vectorization (where for simplicity in the code below, I assume N is divisible by 4).

```
void mat_mul(int N, float *A, float *BT, float *C)
{
   int i, j, k;
   for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
       for (k = 0; k < N; k+=4) {
         t1 = A[i*N+k] * BT[j*N+k];
         t2 = A[i*N+k+1] * BT[j*N+k+1];
         t3 = A[i*N+k+2] * BT[j*N+k+2];
         t4 = A[i*N+k+3] * BT[j*N+k+3];
         C[i*N+j] += t1+t2+t3+t4;
       }
     }
   }
}
```

8220 MFLOPS transpose B and unroll by 4

Coalesce loops

```
void mm_opt(int Ndim, int Mdim, int Pdim, TYPE *A, TYPE *B, TYPE *C){

  #pragma omp parallel for
  for (int it=0; it<Ndim*Mdim; it++){
      int i=it/Mdim;
      int j=it%Mdim;
      TYPE tmp = 0.0;
      for(int k=0;k<Pdim;k++){     /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
           tmp += *(A+(i*Pdim+k)) *  *(B+(k*Mdim+j));
      }
      *(C+(it)) += tmp;
  }
}
```

3270 MFLOPS with 8 threads and matrices of size 300, 600, 900

Many people experimented with different ways to block the matrices … If you block for each level of the cache hierarchy this is how you reach ultimate performance.

But pulling this off in code requires MUCH more time and work on computer architecture

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let’s get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks “out” and leave loops over a single block together

Matrix multiplication: sequential code

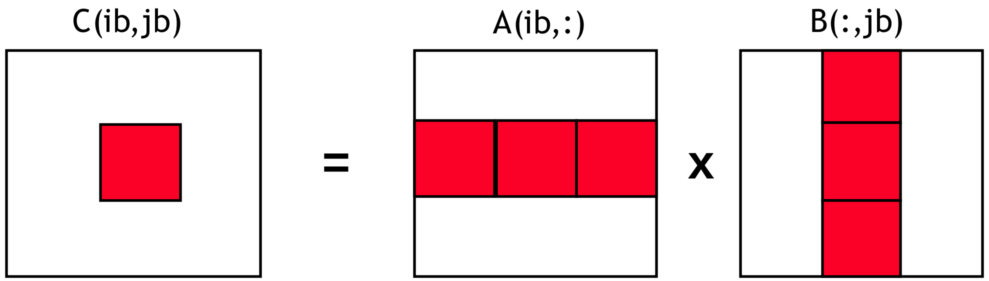
```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
```



```
}
```

A few of you experimented with blocking/tiling … though no one tried the approach I showed here.

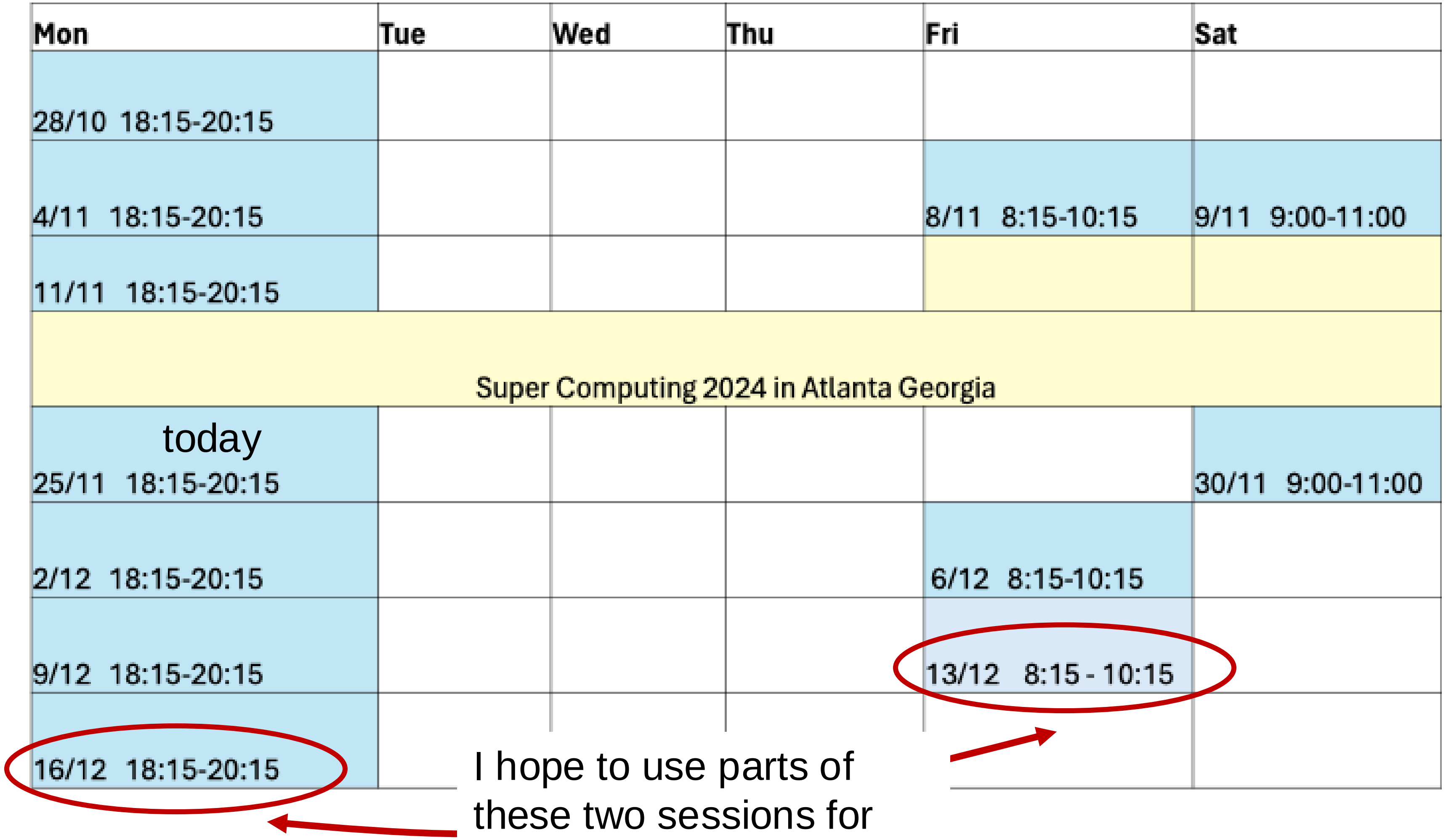
One of you used the ikj algorithm and blocked over k and j … getting one of our best results at 5034 MFLOPS

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Other homework … your class project

- Think about your class project. Remember:
  - This is a five to 10 minute presentation. It’s supposed to be fun and not a HUGE burden.
  - It must be driven by your interests … what about our topic is interesting to you personally.
  - It’s OK to do this alone or as part of a group.

Our remaining lectures



Mon	Tue	Wed	Thu	Fri	Sat
28/10 18:15-20:15					
4/11 18:15-20:15				8/11 8:15-10:15	9/11 9:00-11:00
11/11 18:15-20:15					
Super Computing 2024 in Atlanta Georgia					
today 25/11 18:15-20:15					30/11 9:00-11:00
2/12 18:15-20:15				6/12 8:15-10:15	
9/12 18:15-20:15				13/12 8:15 - 10:15	
16/12 18:15-20:15					

I hope to use parts of these two sessions for your presentations

Examples for the class project

1. Think about a scientific problem YOU REALLY LOVE. Which of the seven dwarfs do people use when working on that problem? Prepare a short description of that work you can share with the class.

2. If you love writing software, implement the Barnes Hutt algorithm. How can you validate that it is correct? Can you parallelize it? Or maybe leave it serial and we'll parallelize it as a class project?

3. If you love computer engineering, pick a novel architecture (such as the Cerebras AI accelerator, https://cerebras.ai/) and create a presentation to summarize it for the class. What (if any) of the computer architecture features we discussed do they use? Do you think it is economically viable?

4. If you love music, pick a popular song and write lyrics that address a topic in HPC or scientific computing. As examples, https://www.youtube.com/watch?v=awjAphkiXwE or https://www.youtube.com/watch?app=desktop&v=whreNeJGCWk or https://www.youtube.com/watch?v=i6rVHr6OwjI

5. Pick a scientific problem you are interested in. What is the system ever simulated on a massive supercomptuer? For example, consider quantum chemistry. What is the largest molecular system ever simulated on an HPC system. Describe it briefly for the class.

6. What is the carbon footprint of global climate model. Maybe if we want to reduce global warming, we should stop simulating it?