

Profiling on HPC Systems

Presented by Nick Hagerty to CSE 6230 at GATech February 14, 2023
Contact: hagertynl@ornl.gov (HAGERTYNL@ORNL.gov)

ORNL is managed by UT-Battelle LLC for the US Department of Energy

Who am I?



Me, June 2021

- Nick Hagerty, BS, MS in Computer Science, Miami (OH) University '21
- Interned at Air Force Research Laboratory in Dayton, OH in computational chemistry 2019-2021
- Joined Oak Ridge National Laboratory in June 2021



Mt LeConte,
GSMNP, TN

What is profiling?

- Gathers information about the time and resources each routine within a program consumes
- Goals:
 - Identify resource-consuming routines to support improving the code base
 - Demonstrate program efficiency
- Available Methods (some of them):
 - Linux – *perf stat*
 - AMD – *rocprof, omniperf*
 - HPE – *perftools*
 - NVIDIA – *nvprof, ncu*
 - Others: HPCToolkit, Apex

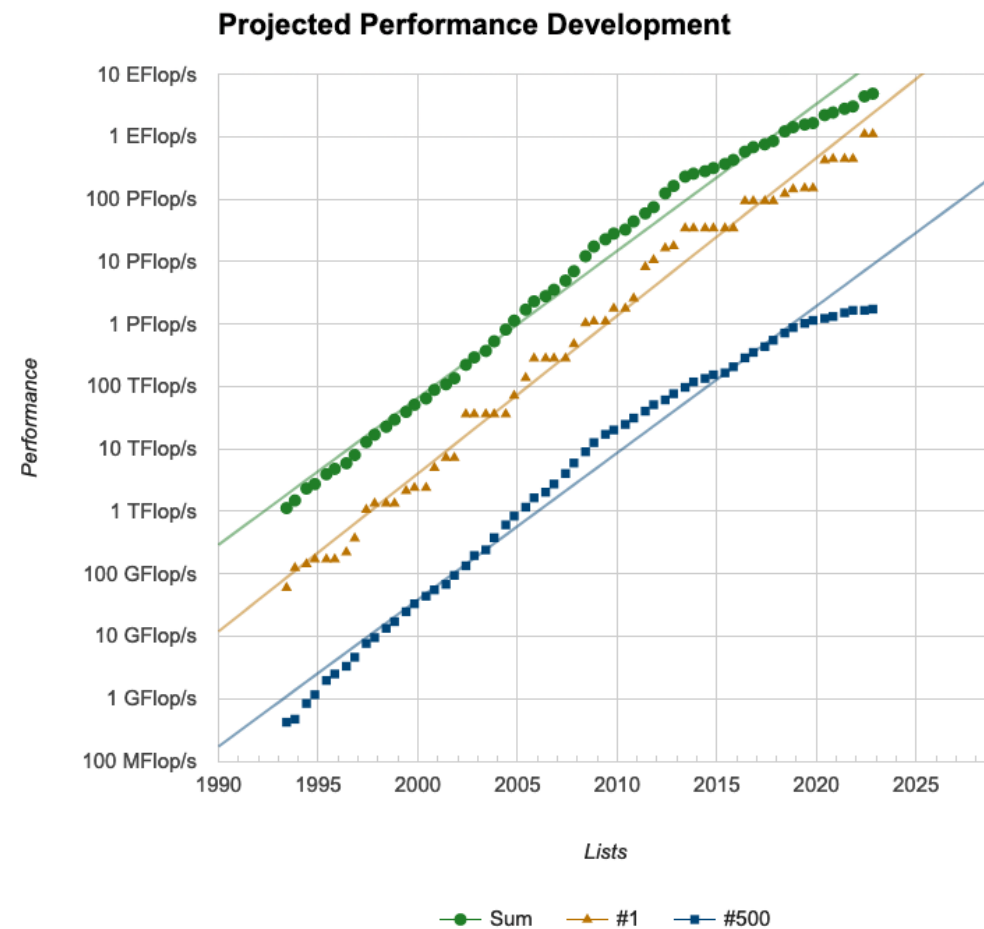
Notation remark:

Flops, flops : floating-point operations

FLOPS, FLOPs : floating-point operations **per second**

Why profile?

- Moore's Law is at a transition
 - Impractical to build larger & larger machines – power, cooling, networking, space
- Application speed-up must rely less on hardware improvement, more on improving algorithms
 - Profiling & optimization

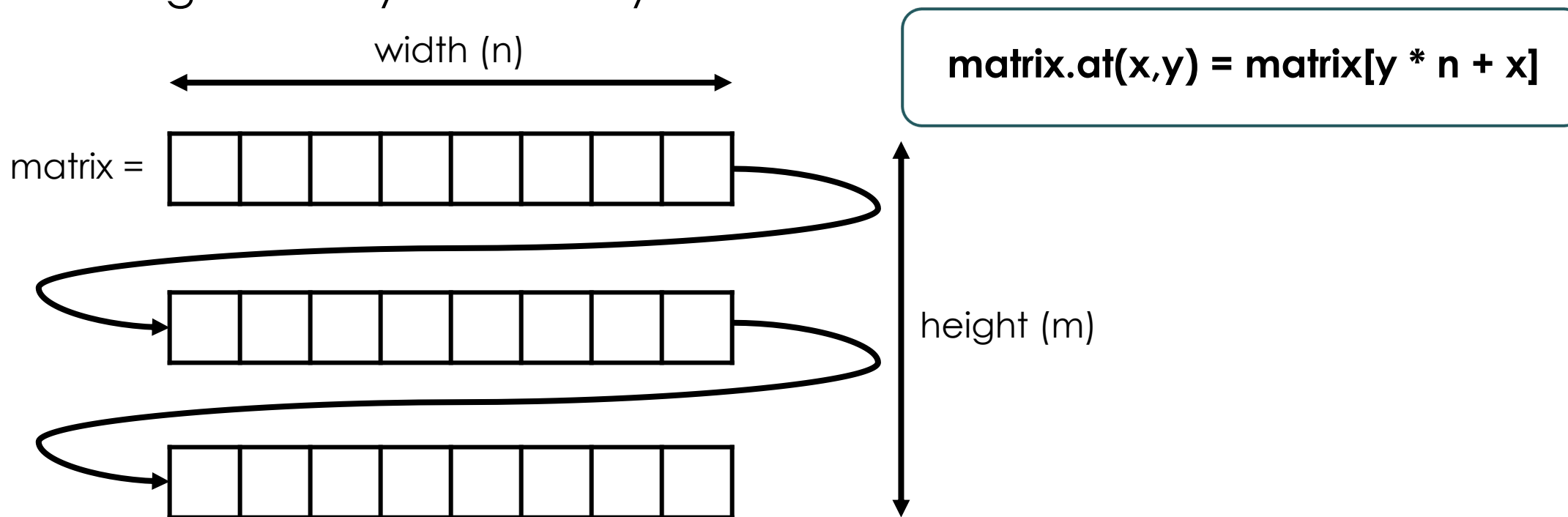


Where we're going with this

- CPU-based profiling: *perf stat*
- GPU-based profiling
 - Introduction: matrix addition
 - Basic Roofline model
 - Ramping up the flops: matrix multiplication
 - Improving matrix multiplication
 - Hierarchical Roofline model
- Demo – GPU stencil

Dipping our toes in – *perf stat*

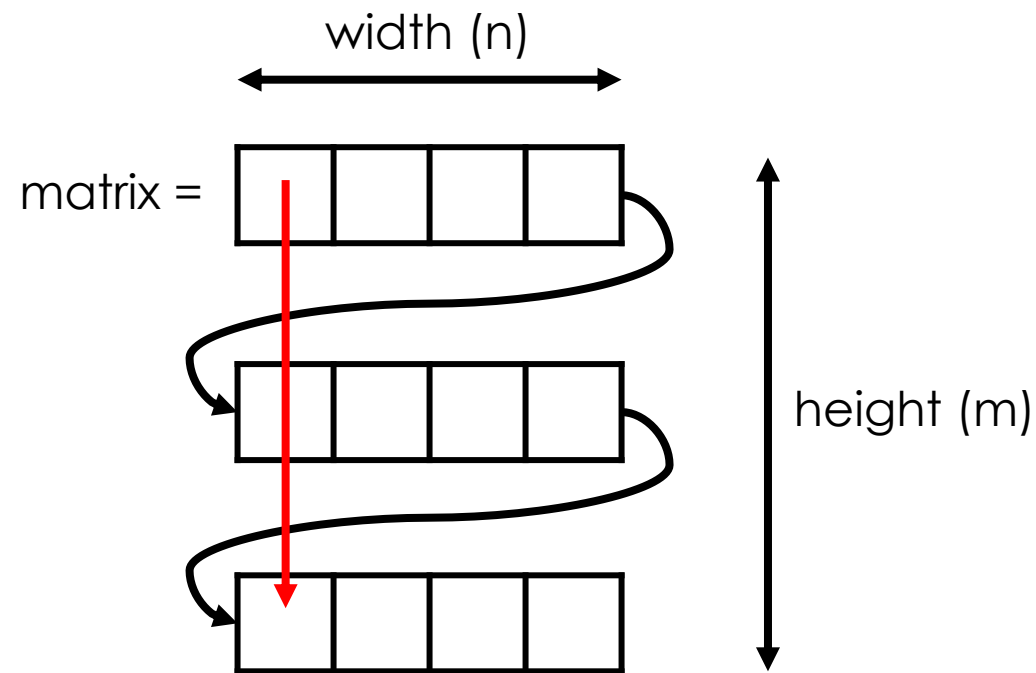
- Linux – *perf stat*
 - Good for basic CPU-based profiling
- Test case: NxN square matrix addition, $C = A+B$
 - Row-major and column-major experiments
- Storing an array in memory:



Dipping our toes in – *perf stat*

- Column-major matrix addition:

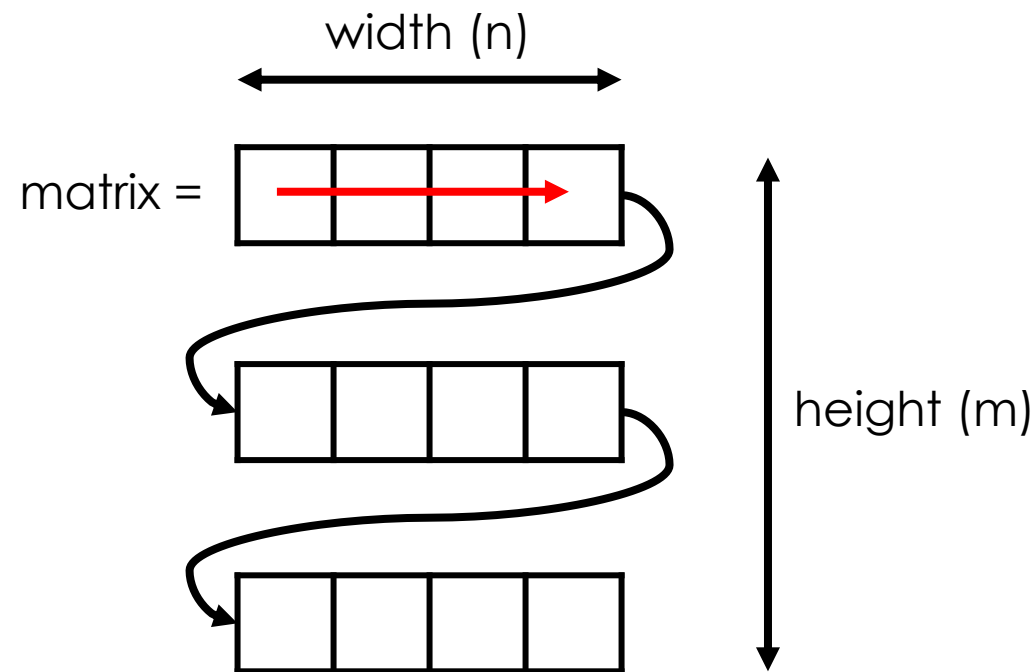
```
for (int i = 0; i < m; i++) {  
    for (int j = 0; j < n; j++) {  
        // inner-most loop changes  
        // which row we select from  
        // ie - move column-by-column  
        C[j * n + i] = A[j * n + i]  
                    + B[j * n + i]  
    }  
}
```



Dipping our toes in – *perf stat*

- Row-major matrix addition:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < m; j++) {  
        // inner-most loop changes  
        // which column we select from  
        // ie - move row-by-row  
        C[i * n + j] = A[i * n + j]  
                    + B[i * n + j]  
    }  
}
```



- Memory reads are typically 64 bytes
 - Reading `matrix[0]` likely reads `matrix[1]`, `matrix[2]`, and `matrix[3]` as well

Dipping our toes in – *perf stat*

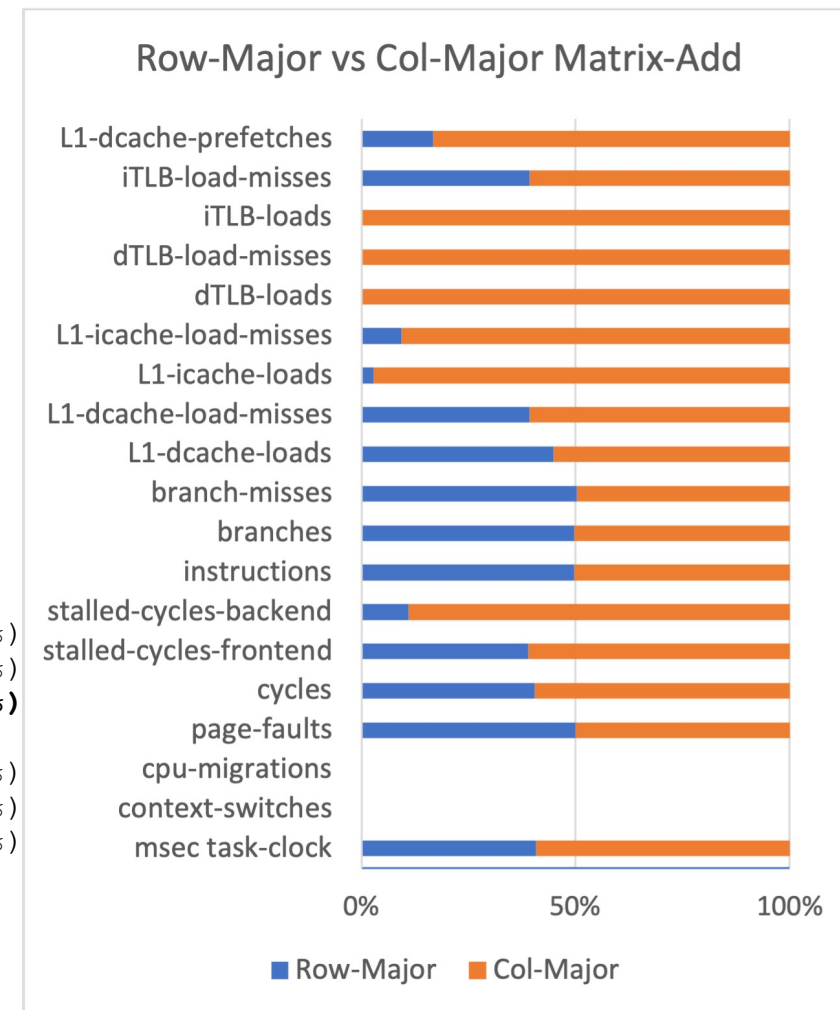
- Linux – *perf stat*
 - Good for basic CPU-based profiling

```
$ perf stat -d -d -d ./matrix-add -m 8192 # These are for column-major
<application output>
Performance counter stats for './matrix-add -m 8192':

 3,691.58 msec task-clock:u          #    1.000 CPUs utilized
          0 context-switches:u       #    0.000 /sec
          0 cpu-migrations:u         #    0.000 /sec
    2,547 page-faults:u              # 689.949 /sec
11,573,436,803 cycles:u              #    3.135 GHz                (83.31%)
 211,027,746 stalled-cycles-frontend:u #  1.82% frontend cycles idle (83.31%)
   7,090,183 stalled-cycles-backend:u #  0.06% backend cycles idle  (83.31%)
11,355,480,740 instructions:u       #    0.98 insn per cycle
          #    0.02 stalled cycles per insn (83.31%)
 2,280,658,033 branches:u           # 617.801 M/sec                (83.38%)
   27,428 branch-misses:u           #    0.00% of all branches    (83.37%)
...many more metrics...

 3.693409405 seconds time elapsed

 3.576172000 seconds user
 0.116005000 seconds sys
```



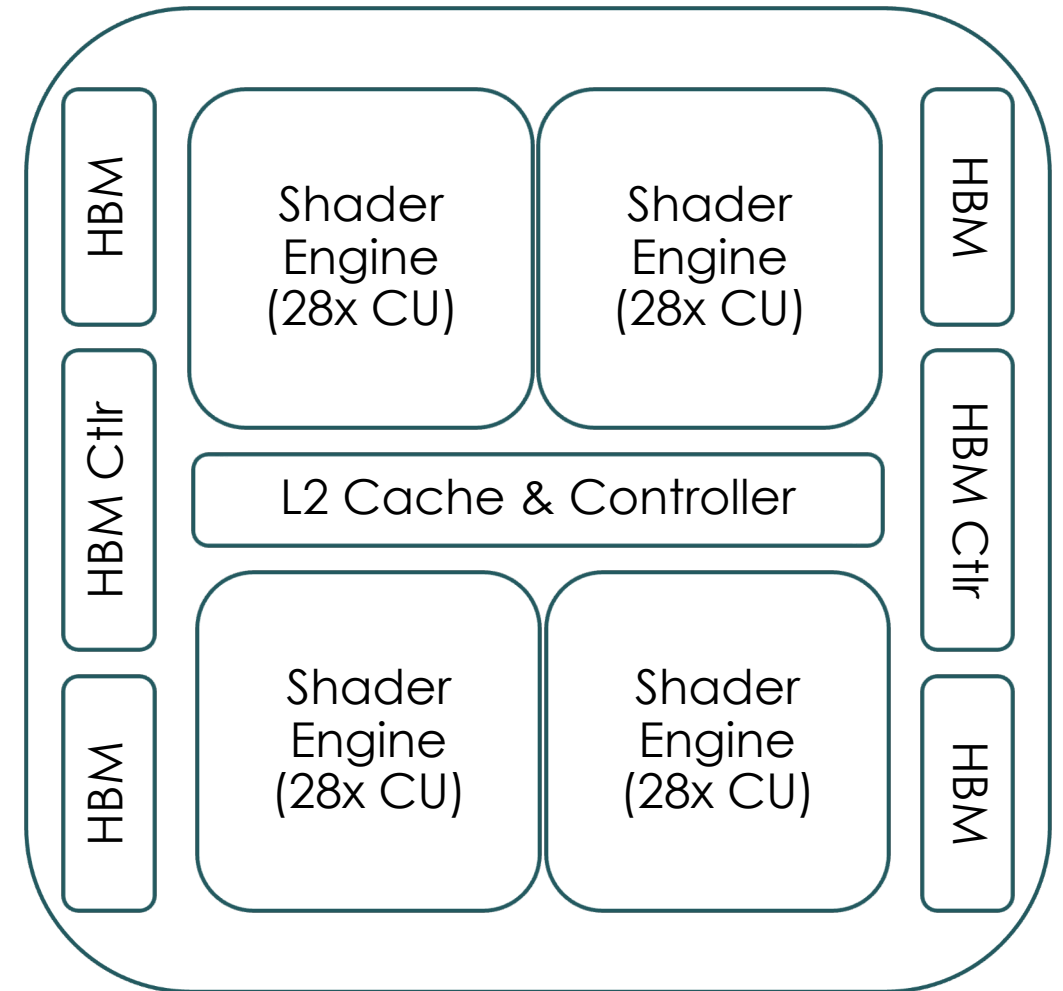
Profiling of row vs column-major matrix addition. 50% indicates identical values for both

Checkpoint 1

- Any questions?
 - Matrix storage in memory
 - Column-major vs row-major addition
 - *perf stat*

The architectures we'll use - GPU

- Profiling in these slides was done using one Graphics Compute Die (GCD) of an AMD MI250X
 - One AMD MI250X contains 2 GCDs, seen as logical GPUs by the runtime environment



Simplified diagram of one AMD MI250X Graphics Compute Die (GCD)

Jumping off the deep end – GPU profiling

- GPU profilers analyze the exact instructions queued and resources consumed by a GPU kernel
- Starting at the bottom – *rocprof* (AMD GPU)
 - Powerful GUI-less primitive profiler
- Each GPU vendor/architecture has a slightly different name for instructions

Operation	AMD MI250X
64-bit floating point addition	SQ_INSTS_VALU_ADD_F64*
32-bit floating point multiply	SQ_INSTS_VALU_MUL_F32*
32-byte read from HBM	TCC_EA_RDREQ_32B_sum**

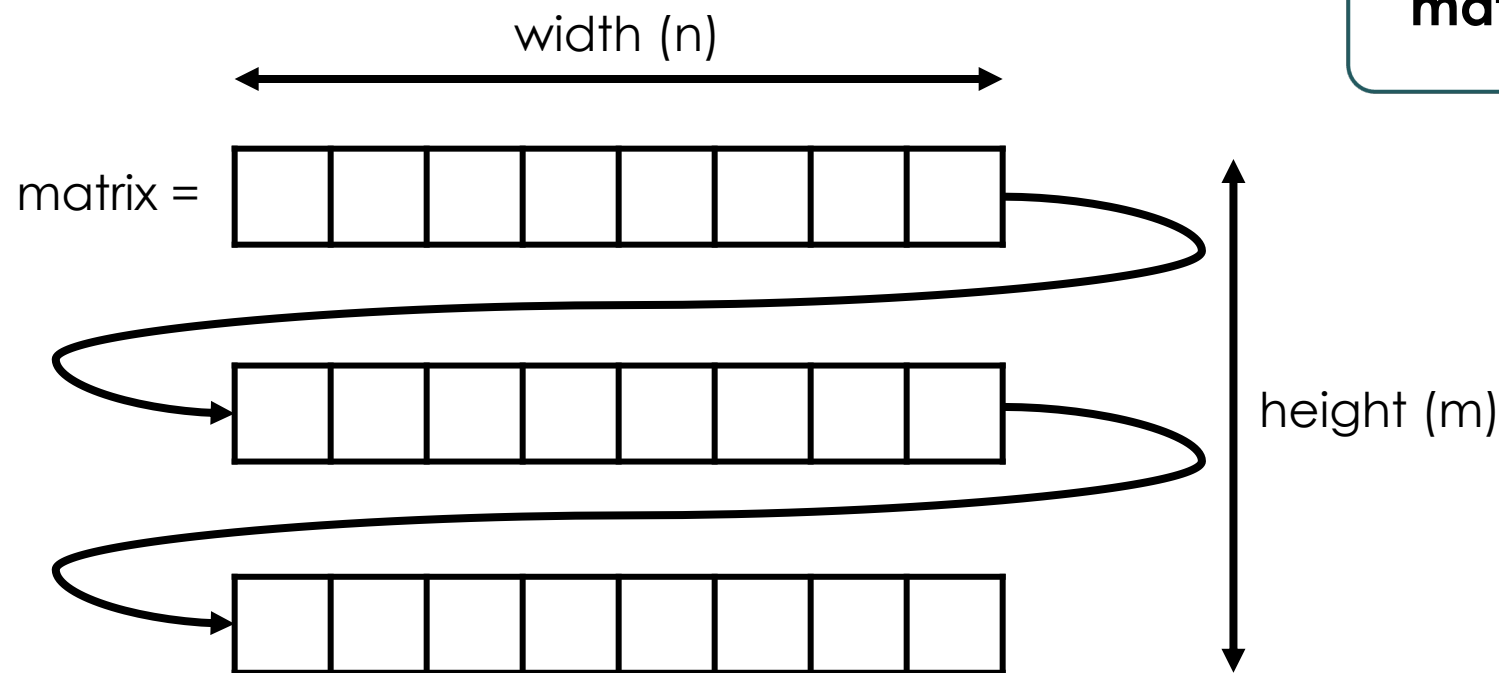
* These instructions are per-wavefront, so they are multiplied by 64

** There are multiple lanes that access HBM, so we sum across these lanes

A basic *rocprof* example – matrix addition

- Matrix addition: $C=A+B$
 - Testing both column-major and row-major
- Storing a matrix in memory:

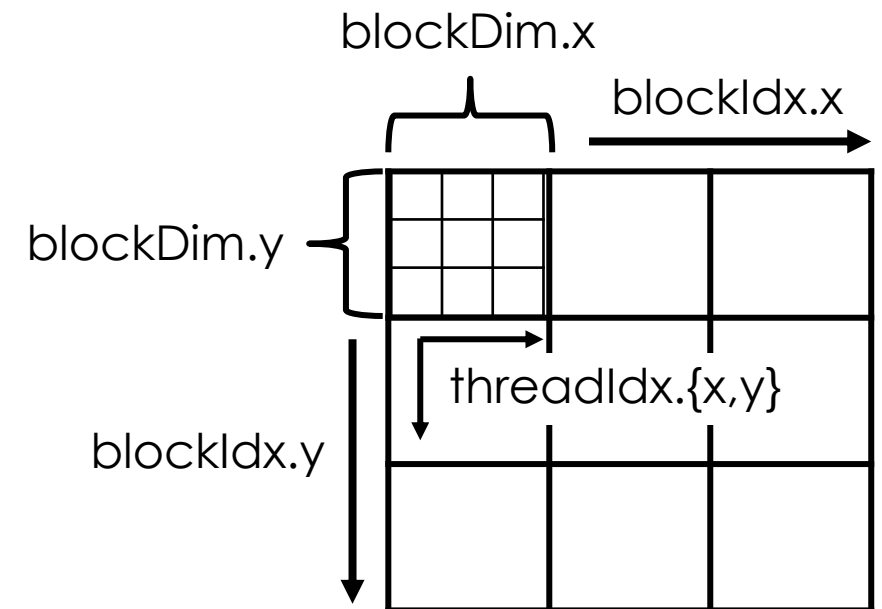
$$\text{matrix.at}(x,y) = \text{matrix}[y * n + x]$$



A basic *rocprof* example – matrix addition

The kernel:

```
// for an n x n square matrix
template<typename T>
__global__ void matrix_add(const T* a, const T* b, T* c, int n,
                           int col_major) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        if (!col_major) {
            int x = row * n + col;
            c[x] = a[x] + b[x];
        } else {
            // then switch row & column
            int x = col * n + row;
            c[x] = a[x] + b[x];
        }
    }
}
```



A basic *rocprof* example – matrix addition

How the kernel is launched for an $n \times n$ matrix:

```
int tpb = 16; // tpb^2 must be < max_threads_per_block (compiler flag)
int block_size = ceil((double) n / (double) tpb);

// Row-major
hipLaunchKernelGGL(matrix_add<T>, dim3(block_size, block_size, 0),
                  dim3(tpb, tpb, 0), 0, 0, a, b, c, n, 0);

// Cow-major
hipLaunchKernelGGL(matrix_add<T>, dim3(block_size, block_size, 0),
                  dim3(tpb, tpb, 0), 0, 0, a, b, c, n, 1);
```

A basic *rocprof* example – matrix addition

- Matrix addition: $C=A+B$
 - Testing both column-major and row-major
- Gather all metrics for HBM and L2 cache, Vector-ALU FP64 usage

Example rocprof input file (line-wrapped for viewing):

```
-----  
pmc : TCC_EA_RDREQ_32B_sum TCC_EA_RDREQ_sum TCC_EA_WRREQ_sum  
      TCC_EA_WRREQ_64B_sum SQ_INSTS_VALU_ADD_F64  
      SQ_INSTS_VALU_MUL_F64 SQ_INSTS_VALU_FMA_F64  
      SQ_INSTS_VALU_TRANS_F64  
pmc : TCC_READ_sum TCC_WRITE_sum  
      TCP_TCC_READ_REQ_sum TCP_TCC_WRITE_REQ_sum  
gpu : 0  
kernel: matrix_add  
-----
```

Each `pmc :` line generates 1 application re-run*.

*When the user wants more metrics than the profiler can handle at once, the application is re-run.

A basic *rocprof* example – matrix addition

Launch command:

```
$ srun -N 1 -n 1 --gpus=1 rocprof -i rocprof.input.txt \  
    --timestamp on -o profile.madd.csv ./matrix_add_gpu
```

Content of profile.madd.csv:

```
Index,KernelName,...,TCC_EA_RDREQ_32B_sum,TCC_EA_RDREQ_sum,...  
2,"void matrix_add<double>(double const*,...)" ,...,0,67114780,...  
<one row for each kernel>
```

- What can we do with these results?
 - With simple kernels, validating that profiling matches expectation
 - If you use a 1 GB matrix, make sure your bytes read is about 2x 1 GB
 - Calculate floating-point performance (Flops per second)
 - Check caching, register pressure, shared memory usage
 - For complex kernels, *roofline* profiling is a good model of performance

A basic *rocprof* example – matrix addition

Validation

Bytes Read from HBM:

ideal read: 2x 4096x4096 matrices of doubles = 268.4 MB // for A & B

```
// compute number of bytes read from HBM
bytes_read = 32 * TCC_EAC_RDREQ_32B_sum +
             64 * (TCC_EA_RDREQ_sum - TCC_EA_RDREQ_32B_sum)
// avg over last 3 kernel invocations:
```

```
bytes_read_rowmajor = 281.7 MB
bytes_read_colmajor = 283.0 MB
```

Flops:

```
flops_fp64 = 64 * (SQ_INSTS_VALU_ADD_F64 + SQ_INSTS_VALU_MUL_F64 +
                  SQ_INSTS_VALU_TRANS_F64 + 2*SQ_INSTS_VALU_FMA_F64)
```

```
time = (CompleteNs - BeginNs) / power(10, 9)
```

```
flops_per_s = flops_fp64 / time
```

```
// avg over last 3 kernel invocations
```

```
flops_per_s_rowmajor = 20.93 GFLOPs
flops_per_s_colmajor = 16.16 GFLOPs
```

A basic *rocprof* example – matrix addition

- What conclusions can we draw from this?
 - Row-major matrix addition performed almost 30% better than column-major, but the bytes read from HBM were very similar – how can we explain this performance difference?

Action items:

1. We should look at the LDS, L1 & L2 cache activity now, since HBM usage doesn't show anything significant
2. We have no idea if 21 GFLOPs is any good on the current hardware
 - a. This is often the case when profiling complex kernels

A basic *rocprof* example – matrix addition

Checking L2 cache operations

Bytes to/from L2 cache controller:

ideal read: 2x 4096x4096 doubles = 268.4 MB // for A & B

ideal write: 1x 4096x4096 doubles = 134.2 MB // for C

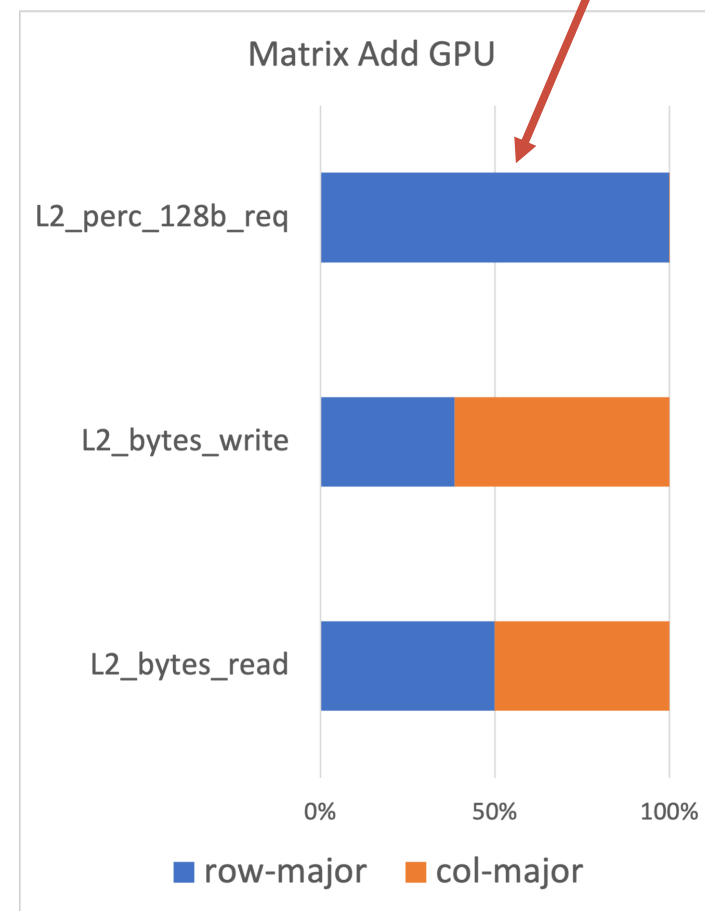
```
// compute number of bytes read/written
bytes_read* = 64 * TCP_TCC_READ_REQ_sum
bytes_write = 64 * TCP_TCC_WRITE_REQ_sum
```

```
// avg over last 3 kernel invocations:
bytes_read_rowmajor = 353.1 MB
bytes_read_colmajor = 353.7 MB
```

```
bytes_write_rowmajor = 176.6 MB
bytes_write_colmajor = 282.5 MB
```

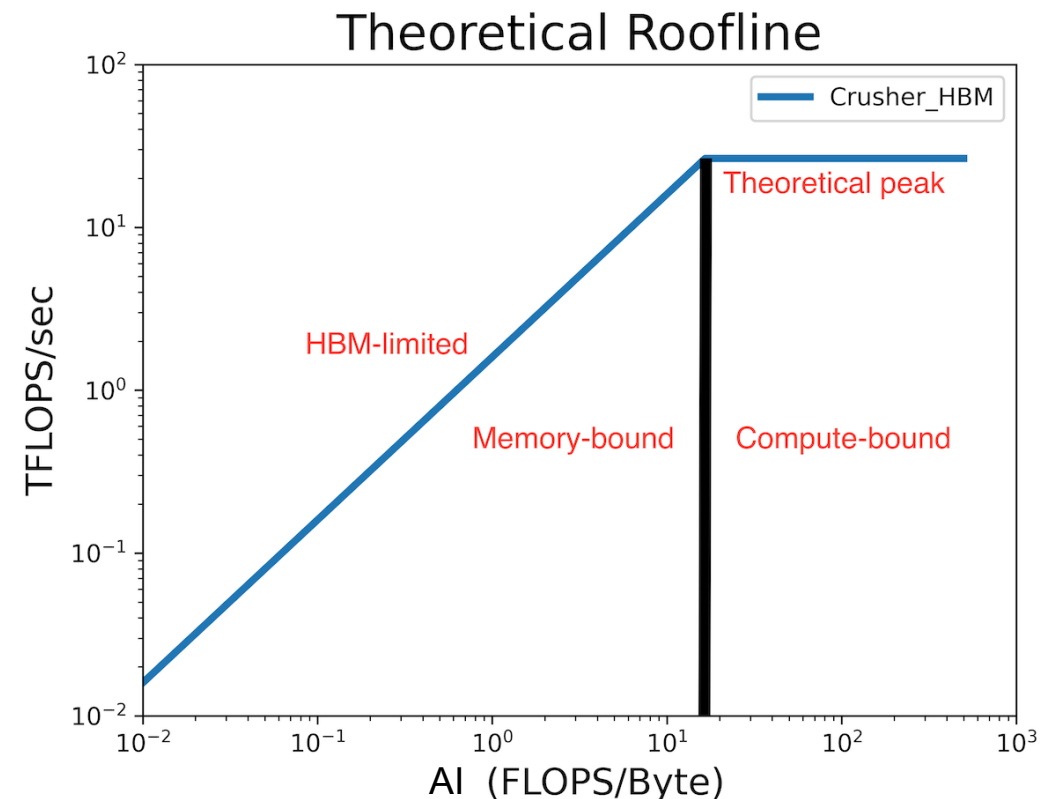
*AMD MI250X can send 2 64-byte reads as one 128-byte read

row-major sends 30% of it's reads in 128-byte chunks, vs 0.01% for column-major



Introduction to Roofline Modeling

- A Roofline model[1] plots floating-point performance as a function of arithmetic (or operational) intensity
 - Your performance is dependent on required bytes from memory/cache
- Example: Crusher – an AMD MI250X-powered test & development system at ORNL[1]



$$\text{FLOPS}_{\text{peak}} = \min(\text{ArithmeticIntensity} * \text{Bandwidth_HBM}, \text{theoretical_peak_flops})$$

- [1] Williams, S., et al. Communications of the ACM, volume 52, pages 65–76, April 2009.
[2] https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

Introduction to Roofline Modeling – matrix addition

```
// compute number of bytes read & written from HBM (excludes caching)
bytes_total = (32 * TCC_EAC_RDREQ_32B_sum +
              64 * (TCC_EA_RDREQ_sum - TCC_EA_RDREQ_32B_sum))
              + (64 * TCC_EA_WRREQ_64B_sum +
              32 * (TCC_EA_WRREQ_sum - TCC_EA_WRREQ_64B_sum))
```

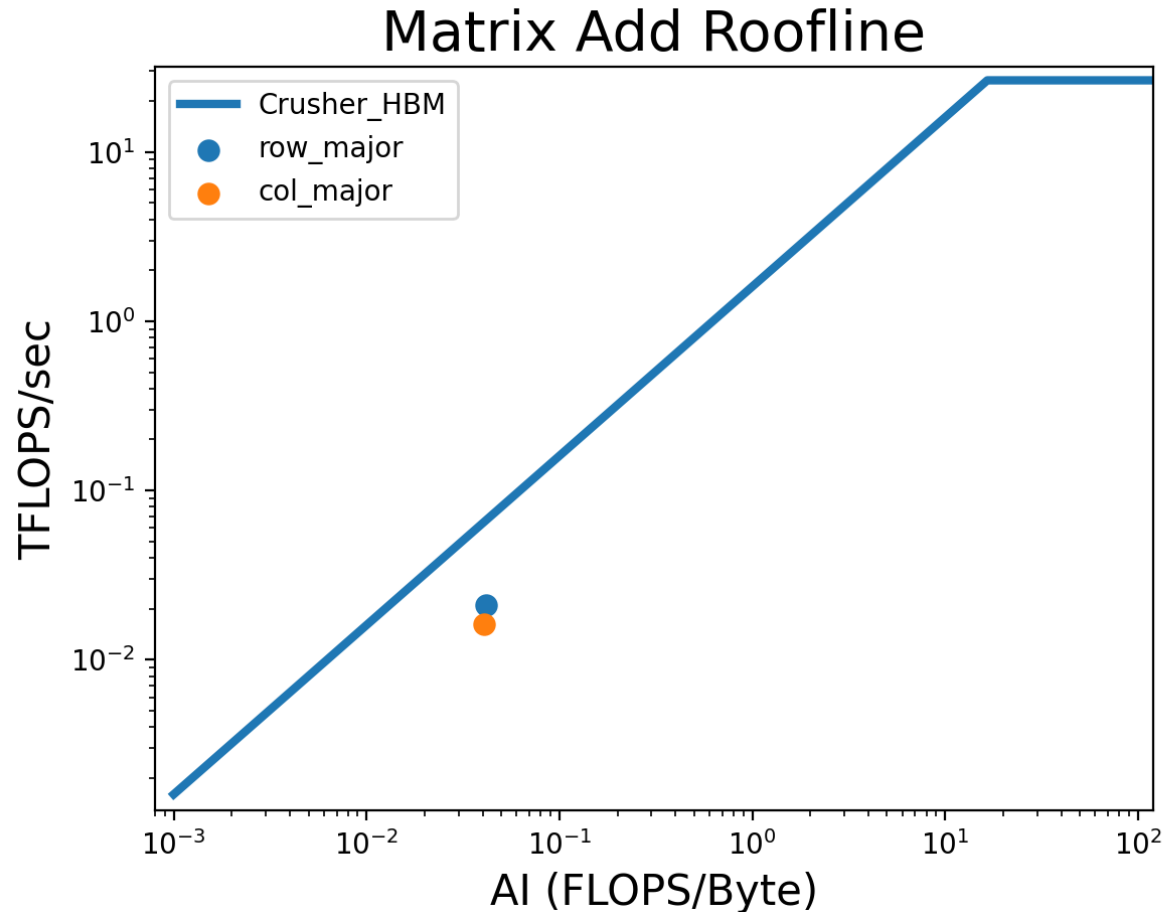
Flops:

```
flops_fp64 = 64 * (SQ_INSTS_VALU_ADD_F64 + SQ_INSTS_VALU_MUL_F64 +
                  SQ_INSTS_VALU_TRANS_F64 + 2*SQ_INSTS_VALU_FMA_F64)
time = (CompleteNs - BeginNs) / power(10, 9)
```

```
ArithmeticIntensity = flops_fp64 / bytes_total
flops_per_s = flops_fp64 / time
```

Alignment	AI (Flops/Byte)	Performance (GFLOPs)	Theoretical Peak (GFLOPs)
Column	0.0408	16.16	65.28
Row	0.0419	20.93	67.04

Introduction to Roofline Modeling – matrix addition



Alignment	AI (Flops/Byte)	Performance (GFLOPs)	Theoretical Peak (GFLOPs)
Column	0.0408	16.16	65.28
Row	0.0419	20.93	67.04

- Achieved about 30% of peak at the given AI for row-major addition
- Theoretical peak determined by formula below:

$$\text{FLOPS_peak} = \min(\text{ArithmeticIntensity} * \text{Bandwidth_HBM}, \text{theoretical_peak_flops})$$

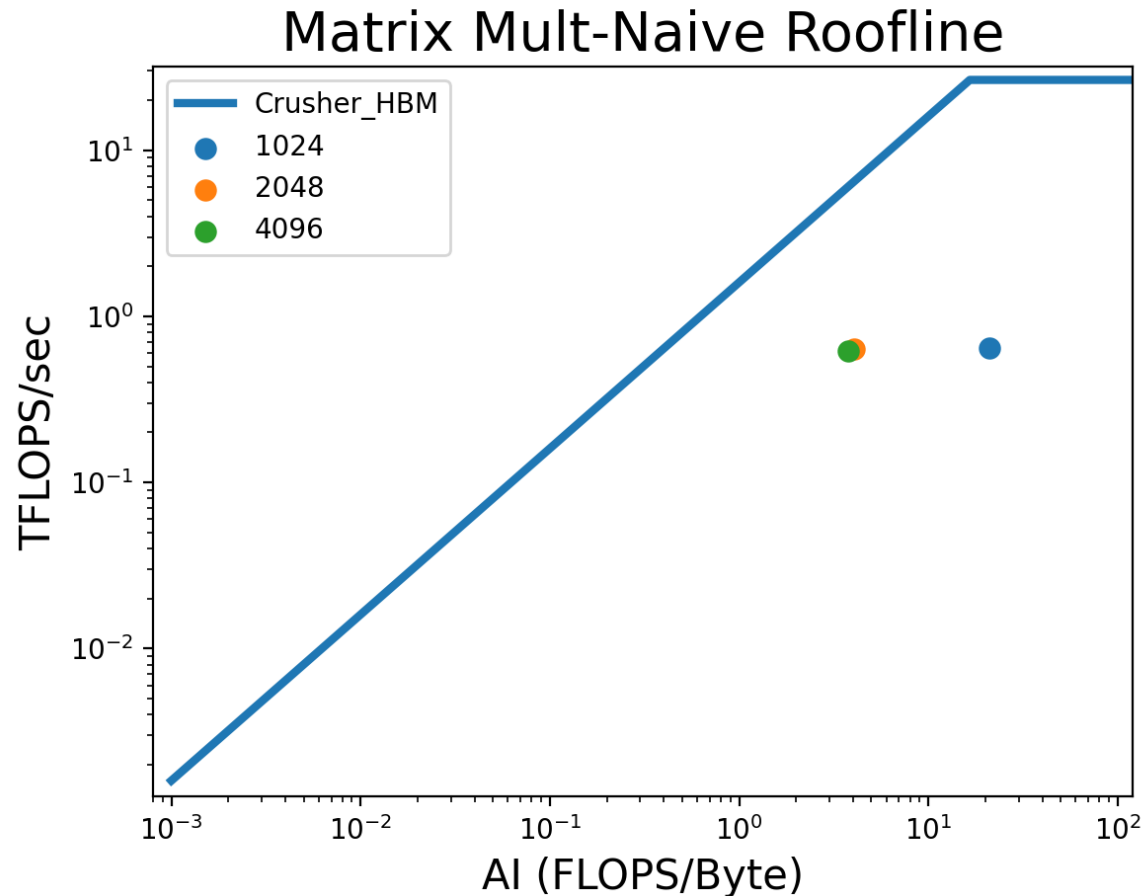
Checkpoint 2

- Any questions?
 - Matrix addition on the GPU
 - rocprof introduction
 - Roofline model introduction

Ramping up the Flops – matrix multiplication

- Matrix addition: $C = Ax + B$
 - If A is $n \times k$ and B is $k \times m$, then:
 - C is $n \times m$, each position in C requires the sum of k multiplications (the dot product of each column of B , row of A)

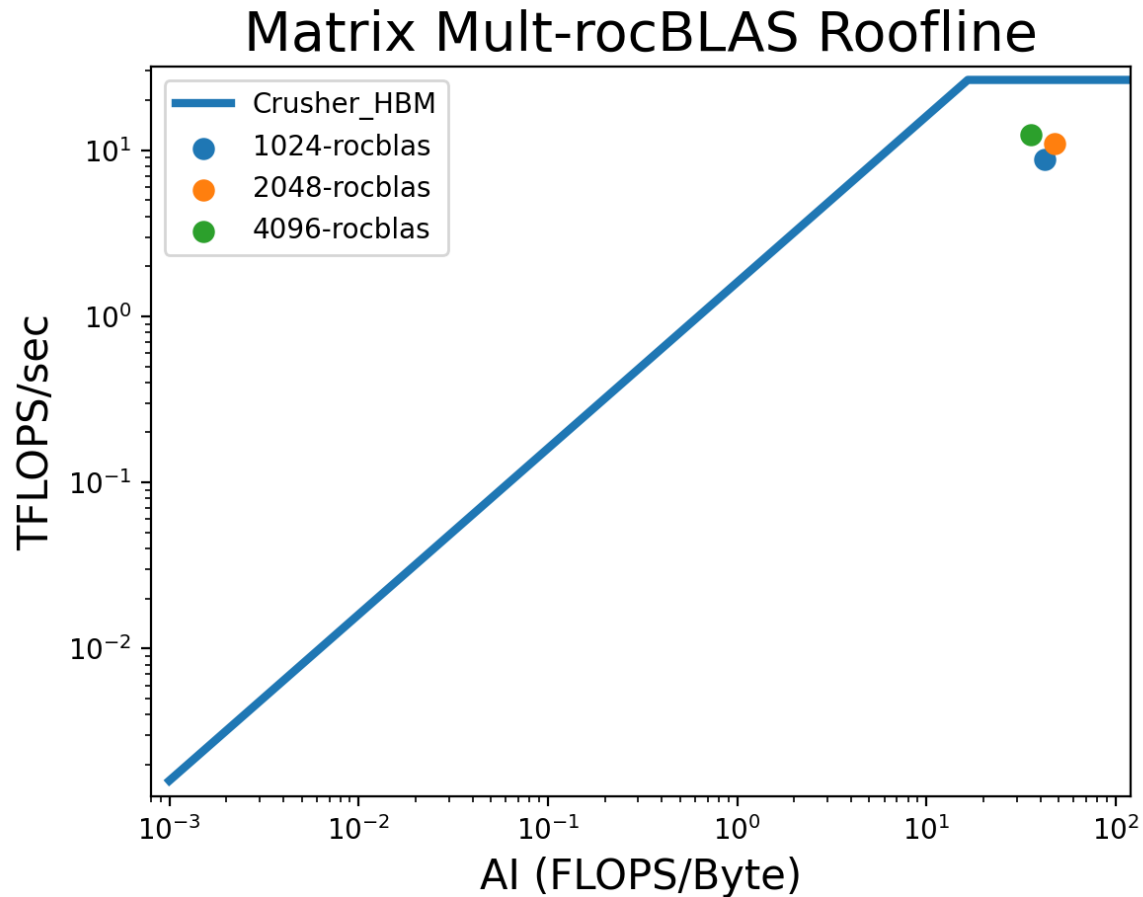
Ramping up the Flops – matrix multiplication



N (N x N matrix)	AI (Flops/Byte)	Performance (GFLOPs)	Roofline Peak (GFLOPs)
1024	21.06	644.63	23900
2048	4.06	631.72	6496
4096	3.78	618.59	6048

- For a matrix multiply, this doesn't look very good – even at larger sizes, achieving <3% of device capability
- Vendors will likely provide libraries for things they want to be highly optimized. Let's try one - **rocBLAS**

Ramping up the Flops – matrix multiplication



N (N x N matrix)	AI (Flops/Byte)	Performance (GFLOPs)	Theoretical Peak (GFLOPs)
1024	42.00	8822.32	23900
2048	47.44	11017.20	23900
4096	35.77	12337.90	23900

- This looks much better
- >10x floating-point performance
- >50% device peak at largest size

Types of Flops – matrix multiplication

- The AMD MI250X provides *Matrix cores*, which are highly optimized to do matrix operations, instead of the standard vector-ALU


Method	ADD_F64*	MUL_F64	FMA_F64	TRANS_F64	MFMA_MOPS_F64	Total GFlops
rocBLAS	0	0	0	0	4194304	1.07**
naive	0	16384	16793600	0	0	2.15

*all metric names above are prefixed with `SQ_INSTS_VALU_` in `rocprof`

**F64 matrix operations are multiplied by 256 to compute the number of performed Flops

At its core, the naïve algorithm has the following line of code to compute a dot-product:

```
value += d_b[r * k + k_idx] * d_a[k_idx * n + c];
```



This is considered a fused multiply-add operation (double the Flops!!)

Types of Flops – matrix multiplication

- The AMD MI250X provides **Matrix cores**, which are highly optimized to do matrix operations, instead of the standard vector-ALU

Method	ADD_F64*	MUL_F64	FMA_F64	TRANS_F64	MFMA_MOPS_F64	Total GFlops
rocBLAS	0	0	0			
naive	0	16384	16793600			

Matrix cores have a higher peak than vector-ALU can achieve – roofline plot needs to be updated

*all metric names above are prefixed with `SQ_INSTS_VALU_` in `rocprof`

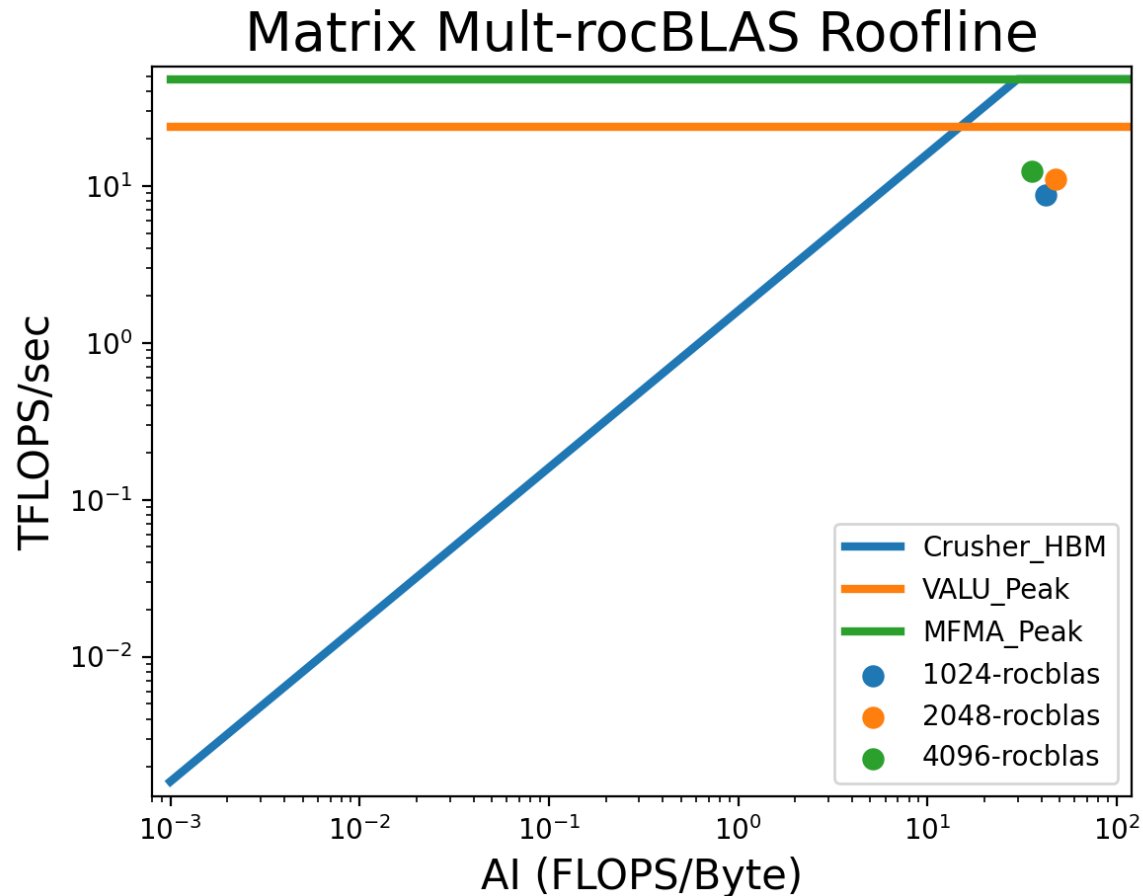
**F64 matrix operations are multiplied by 256 to compute the number of performed Flops

At its core, the naïve algorithm has the following line of code to compute a dot-product:

```
value += d_b[r * k + k_idx] * d_a[k_idx * n + c];
```

This is considered a fused multiply-add operation (double the Flops!!)

Ramping up the Flops – matrix multiplication

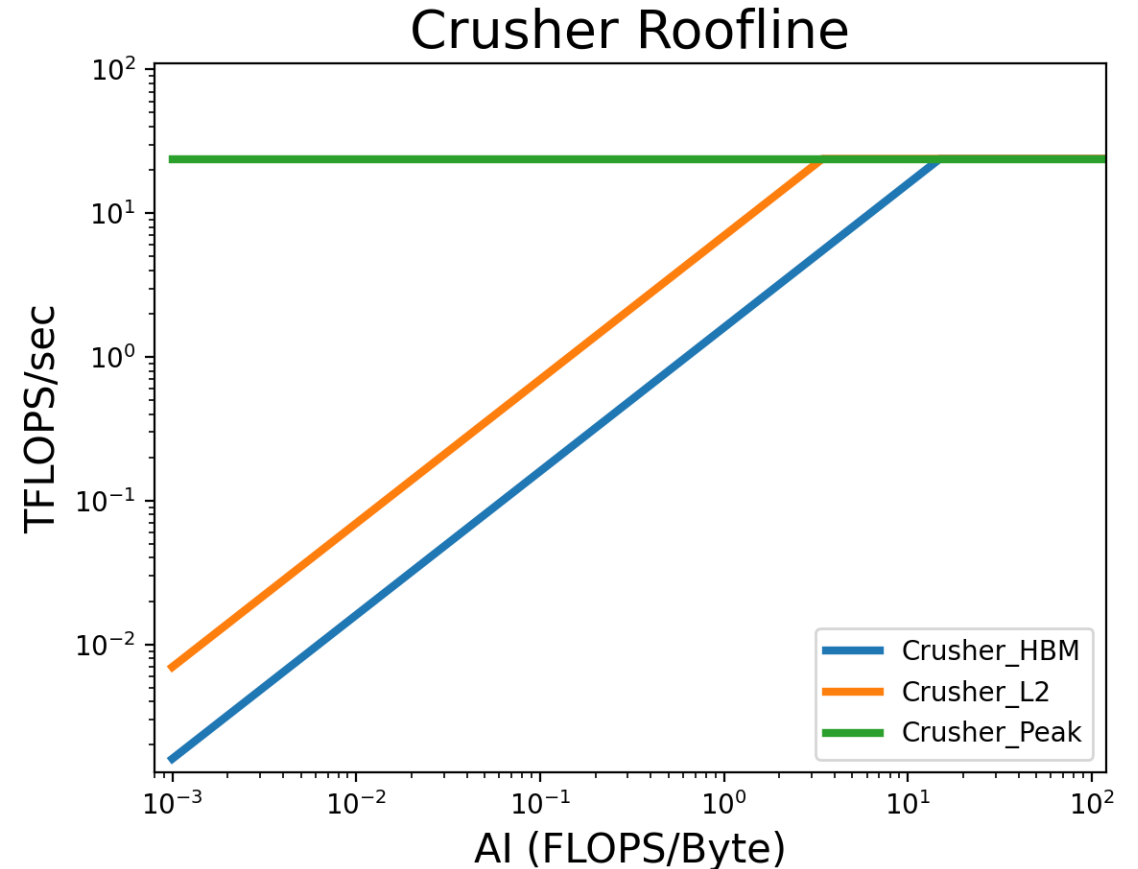


N (N x N matrix)	AI (Flops/Byte)	Performance (TFLOPs)	Theoretical Peak (TFLOPs)
1024	42.00	8.82	23.95 47.9
2048	47.44	11.02	23.95 47.9
4096	35.77	12.34	23.95 47.9

- MI250X provides 2x more theoretical FLOPs for Matrix-FMA instructions

Quick note - Hierarchical Roofline models

- A Hierarchical Roofline model[1] does the same thing as a traditional roofline, but taking into account device caches
- In the rest of this talk, we're going to add the L2 cache in
 - All HBM traffic will pass through the L2 cache, so it is accounted for in that total



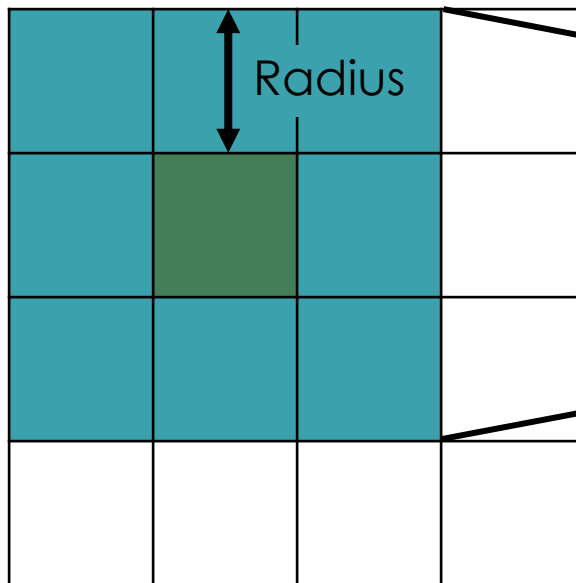
Checkpoint 3

- Any questions?
 - Matrix multiplication on the GPU
 - RocBLAS Matrix multiplication
 - AMD Matrix cores
 - Hierarchical Roofline model

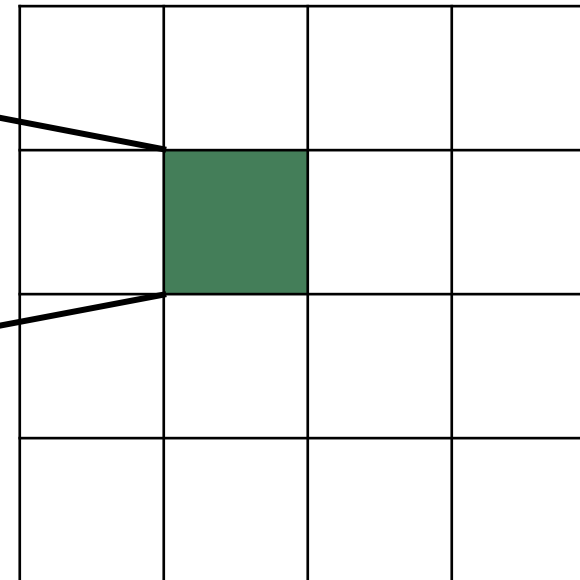
Profiling & Optimization Demo – stencil - image blurring

- 2-D spatial averaging

Input:



Output:



AVERAGE

Demo – stencil - image blurring

- CPU implementation:

```
// ignore edges
for (size_t i = RADIUS + 1; i < n - RADIUS; i++) {
    for (size_t j = RADIUS + 1; j < n - RADIUS; j++) {
        T host_sum = 0;
        for (int i_offset = -RADIUS; i_offset <= RADIUS; i_offset++) {
            for (int j_offset = -RADIUS; j_offset <= RADIUS; j_offset++) {
                host_sum += a_host[(i + i_offset) * n + (j + j_offset)];
            }
        }
        T host_avg = host_sum / ((RADIUS * 2 + 1) * (RADIUS * 2 + 1));
        // use host_avg to check correctness of GPU-generated answer
    }
}
```

Demo – stencil - image blurring

- Naïve GPU implementation:

```
template<typename T>
__global__ void image_blur(T* a, T* b, int n, int m) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row > RADIUS && col > RADIUS && row < m - RADIUS && col < n - RADIUS) {
        for (int i_offset = -RADIUS; i_offset <= RADIUS; i_offset++) {
            for (int j_offset = -RADIUS; j_offset <= RADIUS; j_offset++) {
                b[row * n + col] += a[(row + i_offset) * n + (col + j_offset)];
            }
        }
        b[row * n + col] /= ((RADIUS * 2 + 1) * (RADIUS * 2 + 1));
    }
}
```

Demo – stencil - image blurring

- Naïve GPU implementation:

```
template<typename T>
__global__ void image_blur(T* a, T* b, int n, int m) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row > RADIUS && col > RADIUS && row < m - RADIUS && col < n - RADIUS) {
        for (int i_offset = -RADIUS; i_offset <= RADIUS; i_offset++) {
            for (int j_offset = -RADIUS; j_offset <= RADIUS; j_offset++) {
                b[row * n + col] += a[(row + i_offset) * n + (col + j_offset)];
            }
        }
        b[row * n + col] /= ((RADIUS * 2 + 1) * (RADIUS * 2 + 1));
    }
}
```

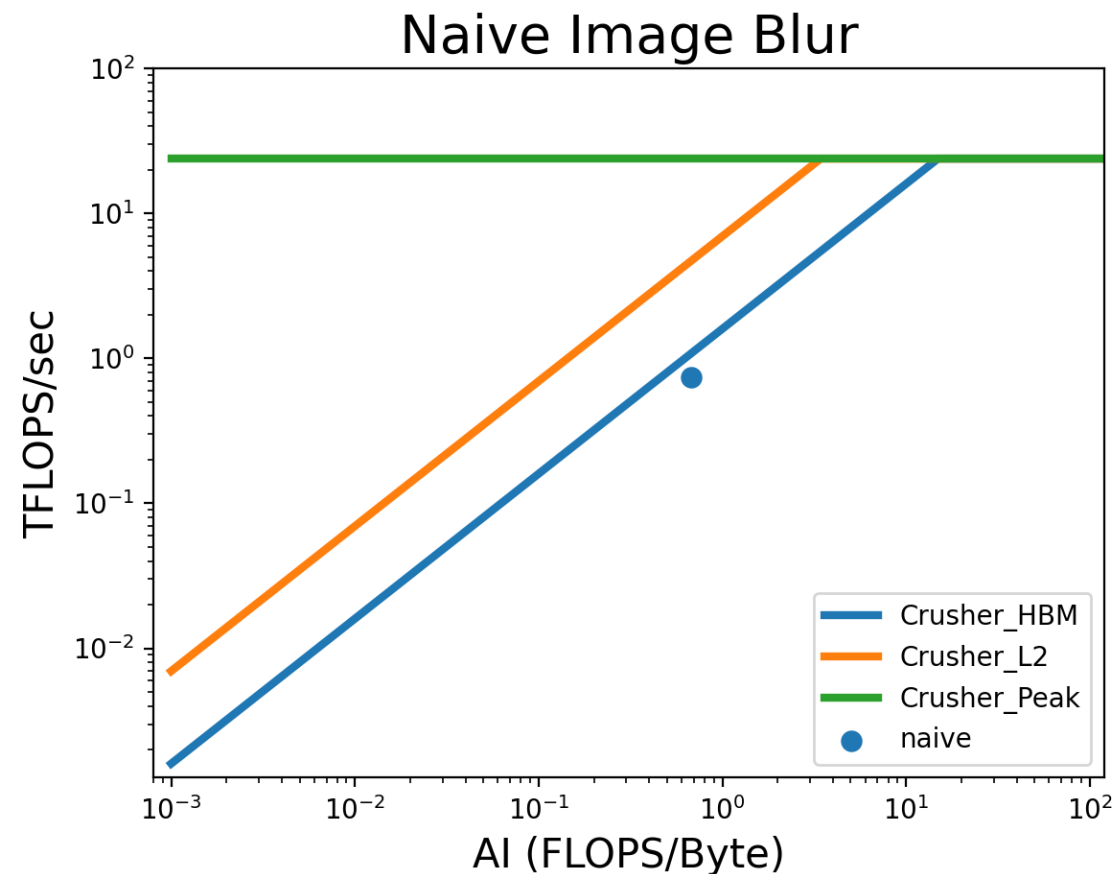
For simplicity, disregard edges

Sum of area

Demo – stencil - image blurring

- Results:

Metric	Value(Naïve)
Bytes_L2	9.13 GB
L2 cache hit (%)	29.4%
AI_L2	0.68
FLOPs	742 GF/s



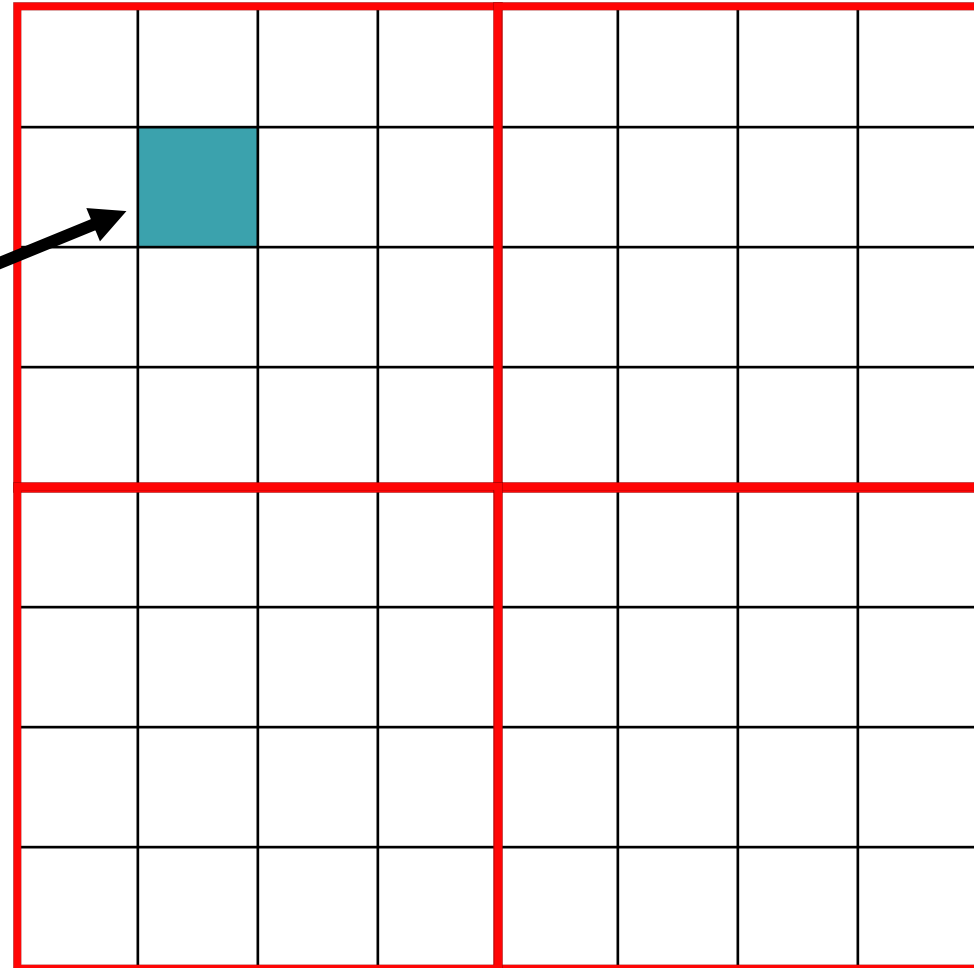
Profiling & Optimization Demo – stencil - image blurring

- Let's expand the picture... Notice how in each block, each space is read multiple times?

Blocks sub-dividing the matrix

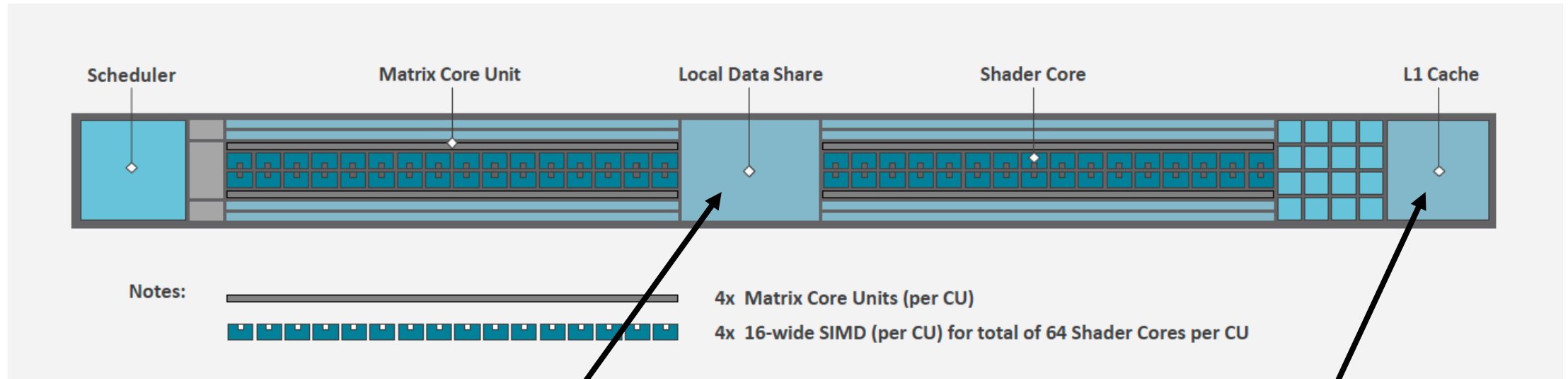
Each space will need to be read by each of its neighbors

If radius == 1, each block is read **9x**



Profiling & Optimization Demo – stencil - image blurring

- Architecture diagram of a single Compute Unit:



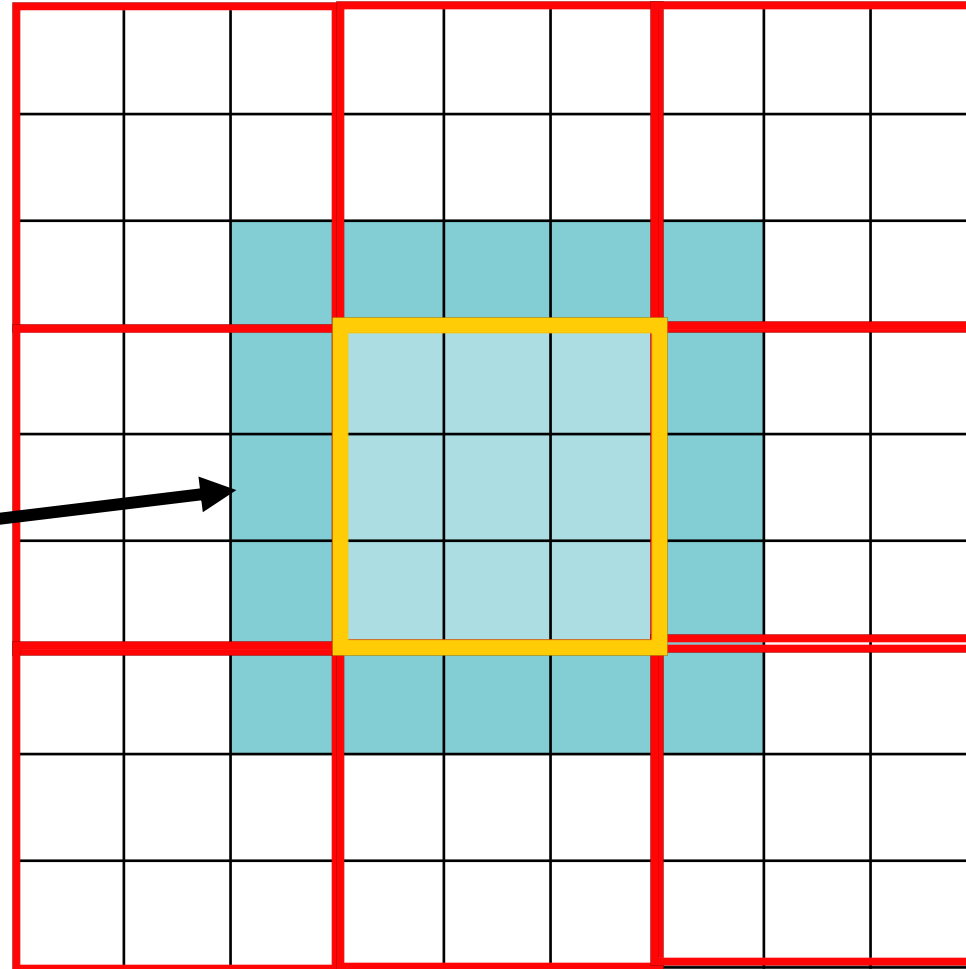
An MI250X contains **64kB** of local data share (LDS), shared by the entire wavefront
- Designed for sharing constant values with other threads across a wavefront

An MI250X contains 16kB of L1 cache, shared by the entire wavefront
- Variables move in & out of L1 cache

Profiling & Optimization Demo – stencil - image blurring

- How we store this in LDS

Load the entries for each block, plus their surrounding neighbors, into matrix in LDS



Demo – stencil - image blurring

- LDS-utilizing implementation:

`__shared__` requires dimensions known at compile-time (`#define`)

```
__global__ void image_blur(T* a, T* b, int n, int m) {  
    __shared__ T a_lds[(THR_PER_BLOCK_X + 2 * RADIUS) * (THR_PER_BLOCK_Y + 2 * RADIUS)],  
                b_lds[THR_PER_BLOCK_X * THR_PER_BLOCK_Y];  
    // omitted - load shared data - ~50 lines of code to cover edge cases  
    if (row >= RADIUS && col >= RADIUS && row < m - RADIUS && col < n - RADIUS) {  
        b_lds[threadIdx.y * THR_PER_BLOCK_X + threadIdx.x] = 0.0;  
        for (int i_offset = -RADIUS; i_offset <= RADIUS; i_offset++) {  
            for (int j_offset = -RADIUS; j_offset <= RADIUS; j_offset++) {  
                b_lds[threadIdx.y * THR_PER_BLOCK_X + threadIdx.x] +=  
                    a_lds[(threadIdx.y + RADIUS + i_offset) * a_lds_dimx  
                        + (threadIdx.x + RADIUS + j_offset)];  
            }  
        }  
        b[row * n + col] = b_lds[threadIdx.y * THR_PER_BLOCK_X + threadIdx.x] / (T) num_squares;  
    }  
}
```

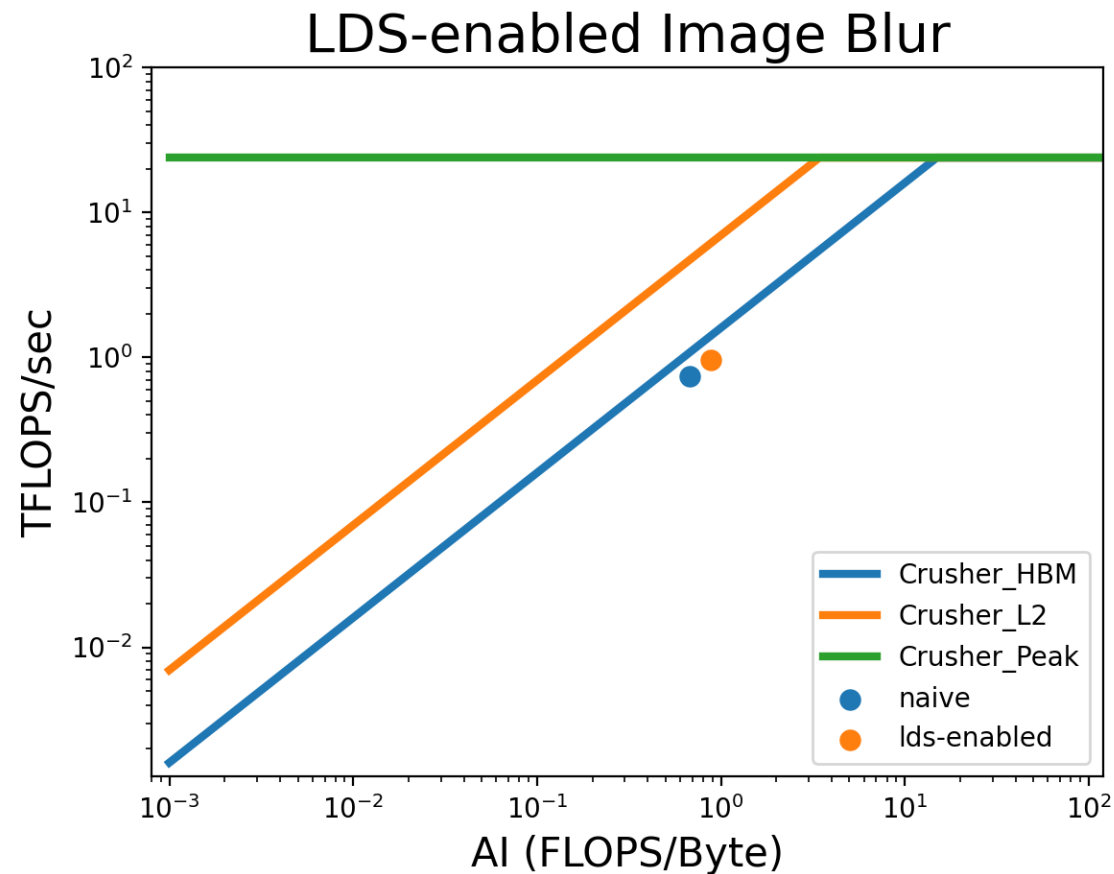
block-local
indexing

*keywords like `__restrict__` and `const` are removed for viewing

Demo – stencil - image blurring

- Result:
 - Increased Arithmetic Intensity
 - 23% fewer bytes requested from L2 cache
 - Increased Flop rate
 - **Exact same number of Flops**

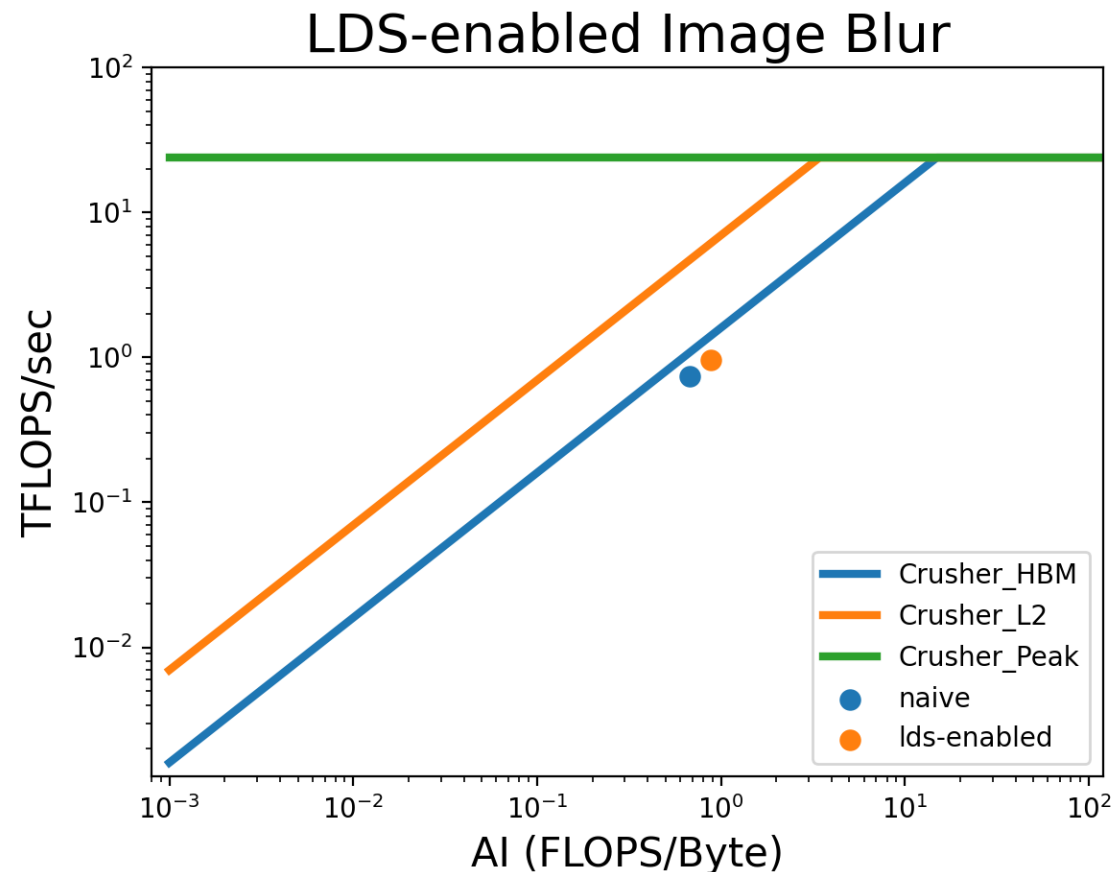
Metric	Naïve	LDS-enabled
Bytes_L2	9.13 GB	6.98 GB
L2 cache hit (%)	29.4%	38.4%
AI_L2	0.68	0.89
FLOPs	742 GF/s	955 GF/s
Bytes_LDS	0 B	5120 B



Demo – stencil - image blurring

- Result:
 - Increased Arithmetic Intensity
 - 23% fewer bytes requested from L2 cache
 - Increased Flop rate
 - **Exact same number of Flops**

Metric	Naïve	LDS-enabled
Bytes_L2	9.13 GB	6.98 GB
L2 cache hit (%)	29.4%	38.4%
AI_L2	0.68	0.89
FLOPs	742 GF/s	955 GF/s
Bytes_LDS	0 B	5120 B

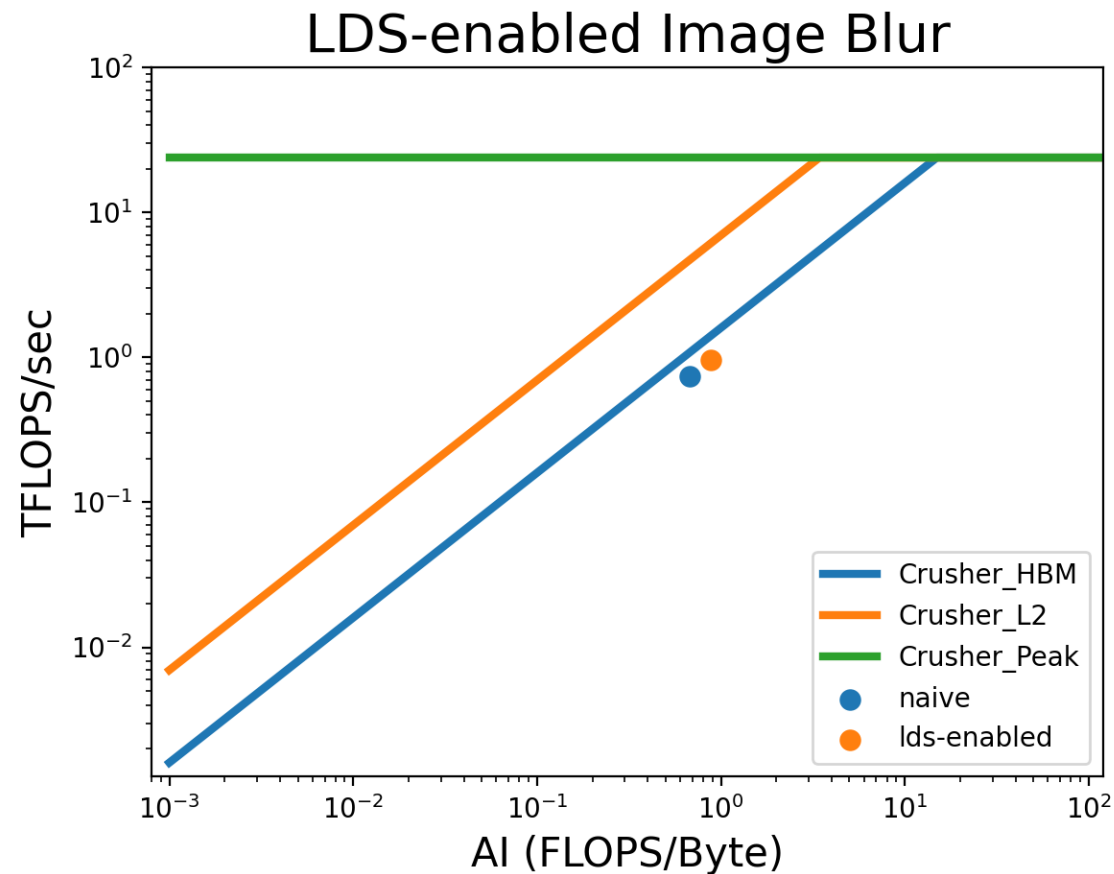


By sending 5KB to LDS, we saved 2GB of reads to L2

Demo – stencil - image blurring

- Result:
 - Increased Arithmetic Intensity
 - 23% fewer bytes requested from L2 cache
 - Increased Flop rate
 - **Exact same number of Flops**

Metric	Naïve	LDS-enabled
Bytes_L2	9.13 GB	6.98 GB
L2 cache hit (%)	29.4%	38.4%
AI_L2	0.68	0.89
FLOPs	742 GF/s	955 GF/s
Bytes_LDS	0 B	5120 B

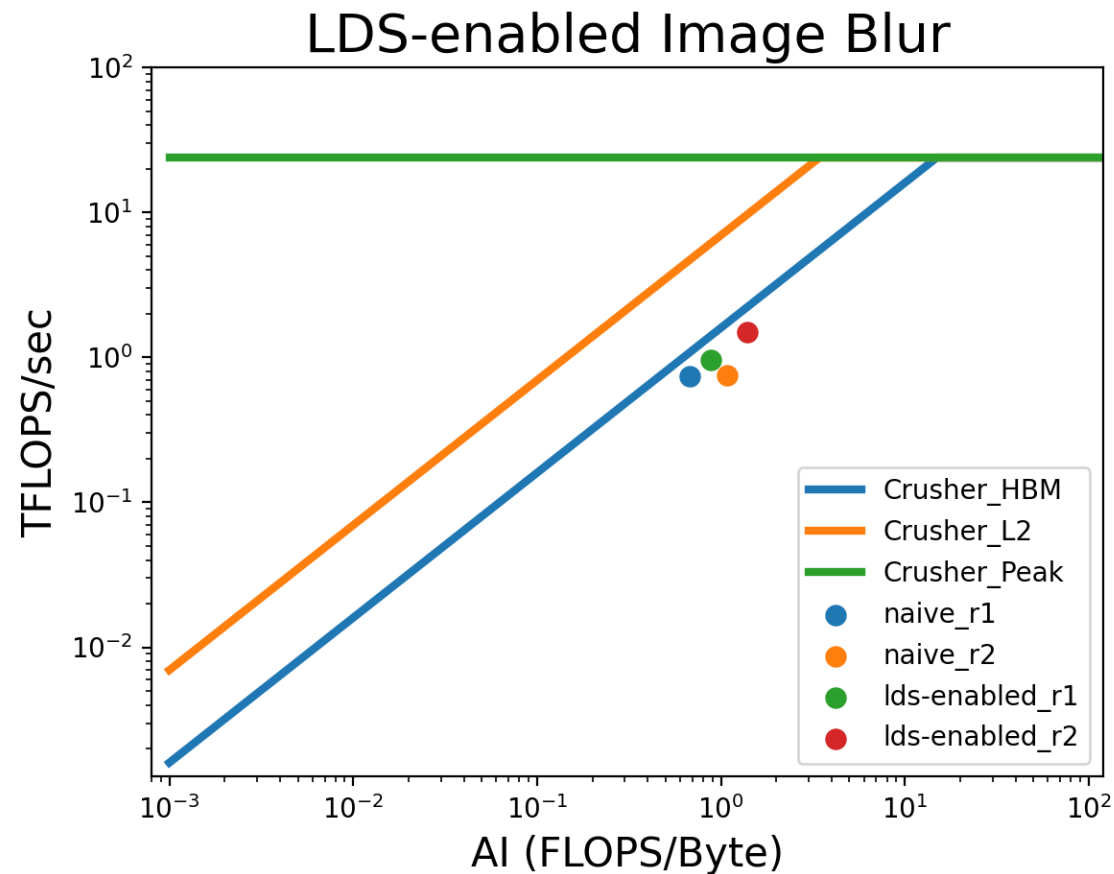


What happens if we increase the radius?

Demo – stencil - image blurring

- Result:
 - Higher AI for both naïve and LDS-enabled
 - No performance gain for naïve kernel, drastically improved performance (+57%) for LDS-enabled kernel

Metric	LDS R=1	LDS R=2
Bytes_L2	6.98 GB	7.52 GB
L2 cache hit (%)	38.4%	42.6%
AI_L2	0.89 Flops/B	1.39 Flops/B
FLOPs	955 GF/s	1498 GF/s
Bytes_LDS	5120 B	5632 B



Outcomes

- Some helpful tips to keep in mind:
 - Know your hardware!
 - Utilize LDS, matrix cores, whatever your architecture can provide you. Matrix cores may be specific to AMD, but LDS is common
 - Mind your memory
 - Minimizing off-chip (HBM) reads can help improve performance, but doesn't always tell the full picture. Feel free to check L2 cache as well. Try to use algorithms that take advantage of the storage of the data structure
 - Reduce the Flops
 - When possible, use algebra to simplify the math required (we didn't cover an example of this today)

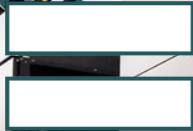
Internships & Jobs

- Pathways to Computing Internship:
 - <https://education.ornl.gov/pathways/>
- Find open job postings:
 - <https://jobs.ornl.gov>

Frontier fun facts



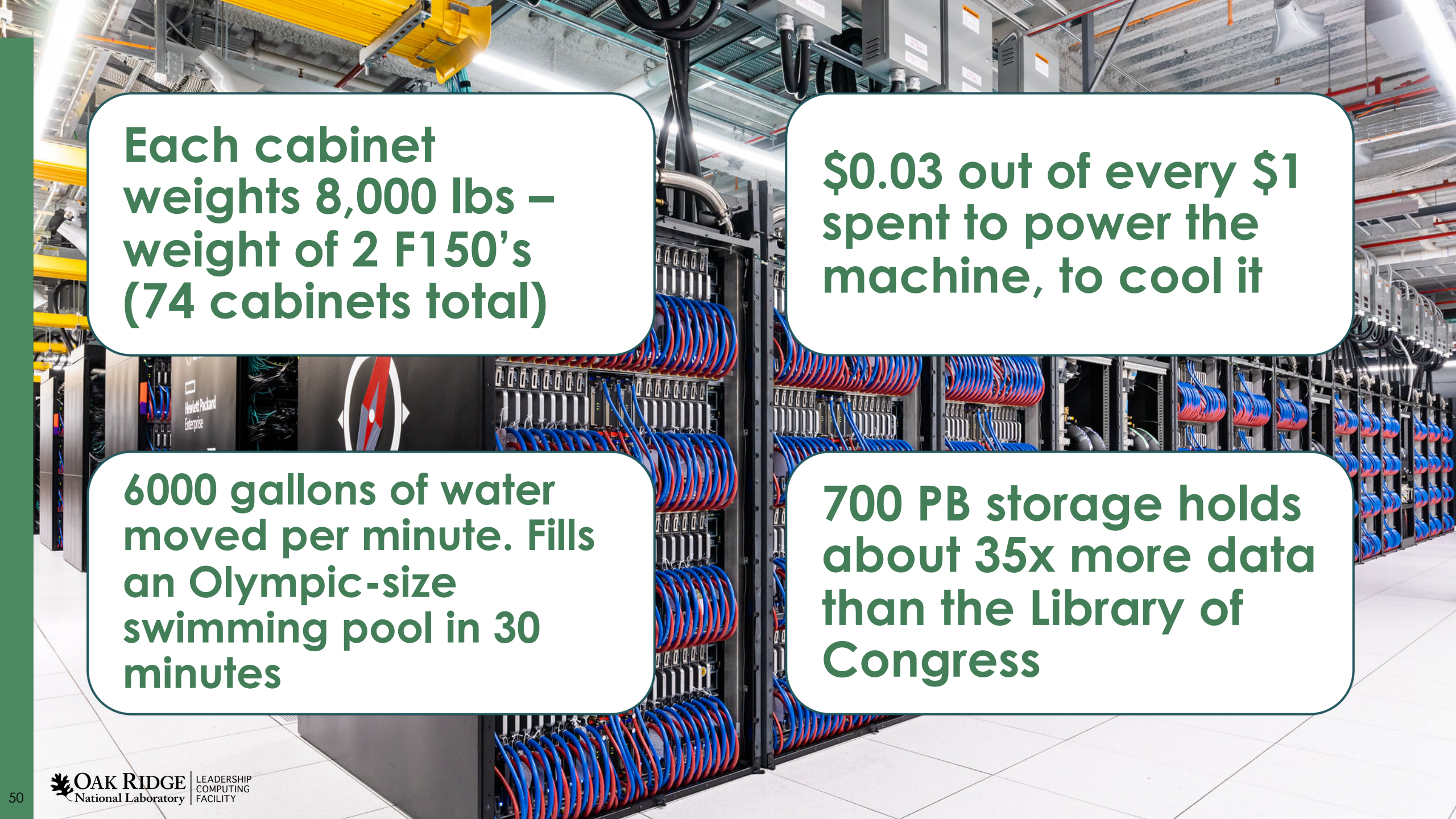
1.1 Exa-FLOPS (floating-point operations)



1 quintillion (10^{18}) calculations per second

which
takes

4 years, if everyone on the earth did 1 calculation per second



Each cabinet
weights 8,000 lbs –
weight of 2 F150's
(74 cabinets total)

\$0.03 out of every \$1
spent to power the
machine, to cool it

6000 gallons of water
moved per minute. Fills
an Olympic-size
swimming pool in 30
minutes

700 PB storage holds
about 35x more data
than the Library of
Congress