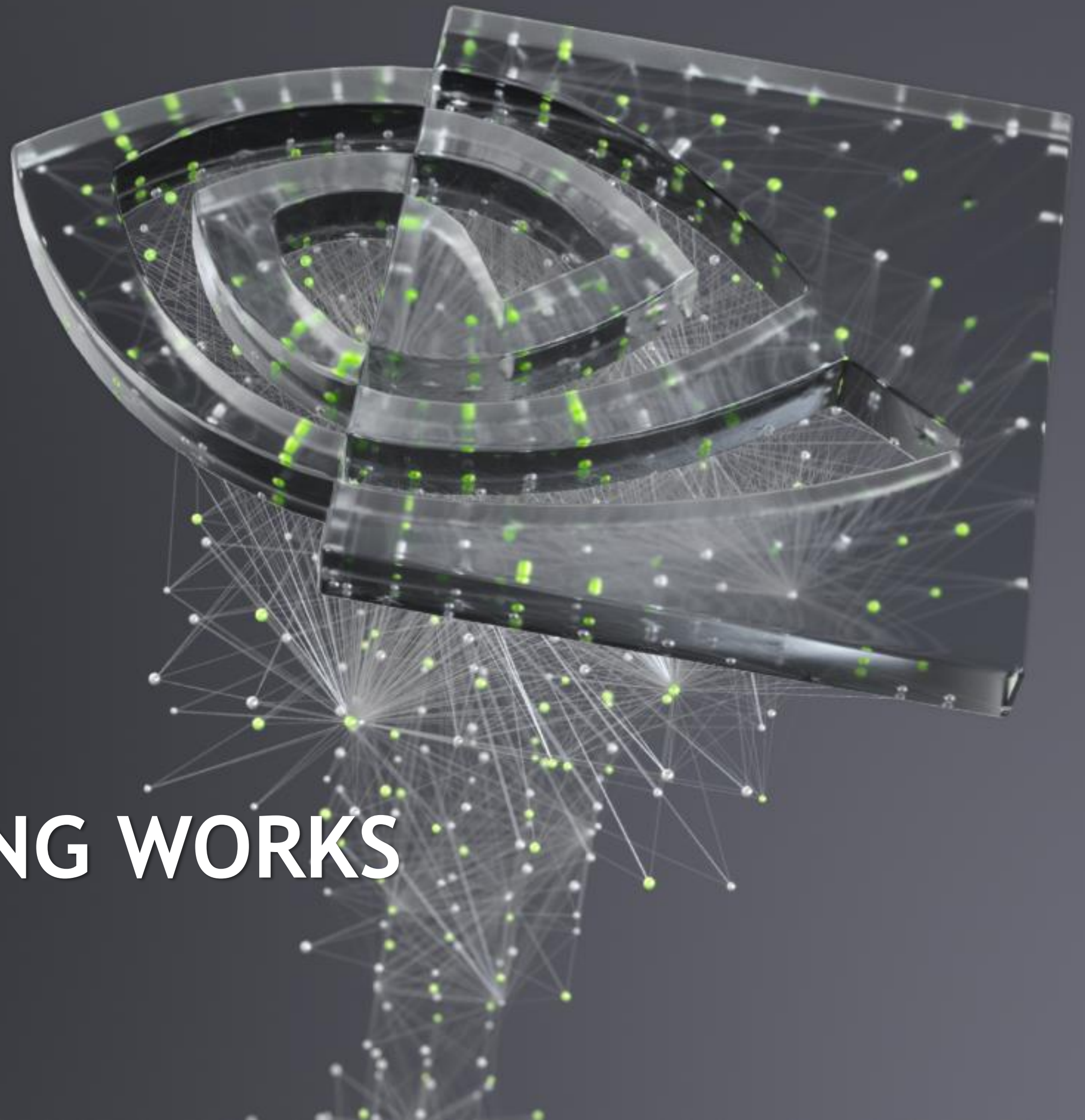


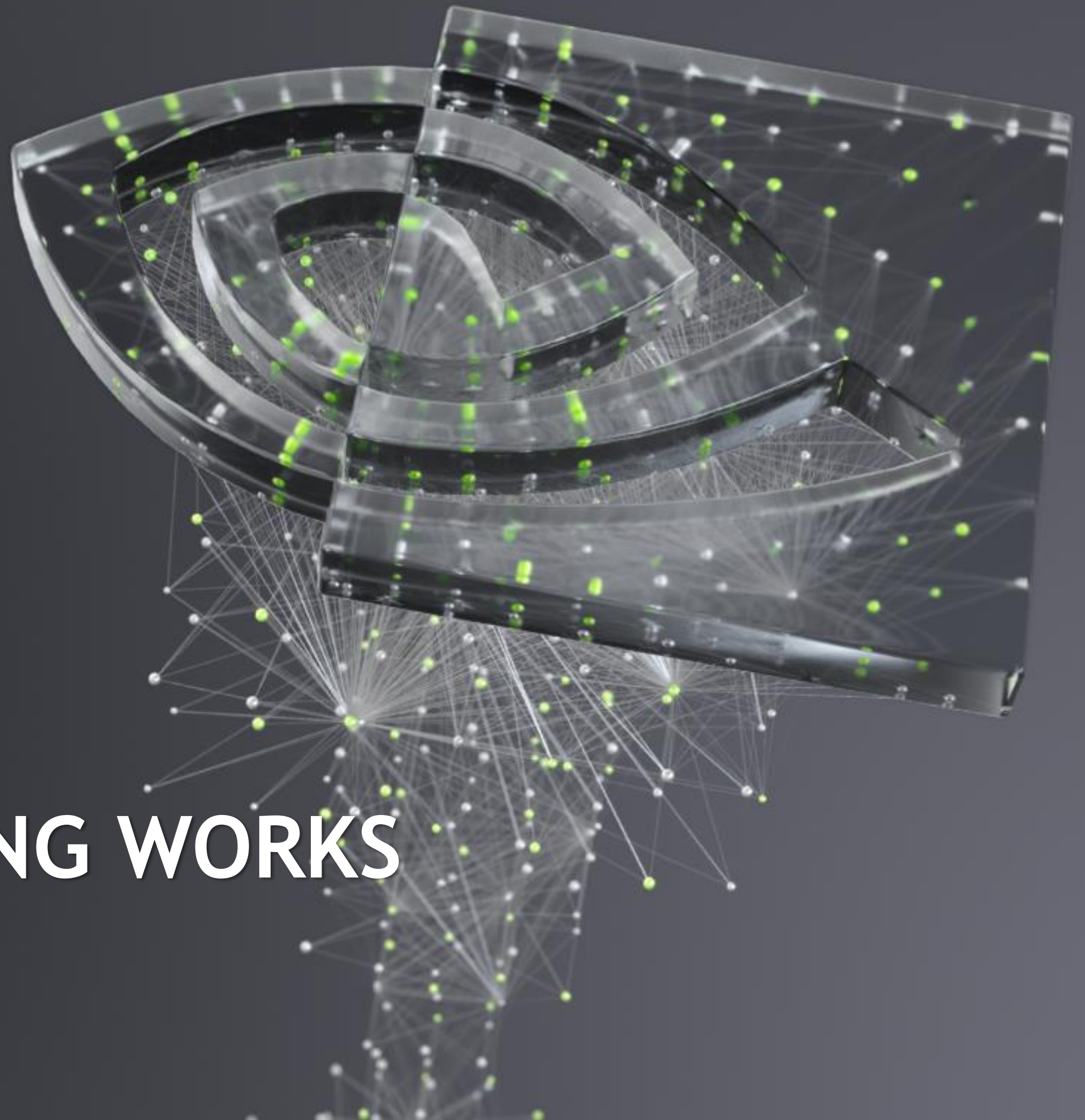


# HOW GPU COMPUTING WORKS

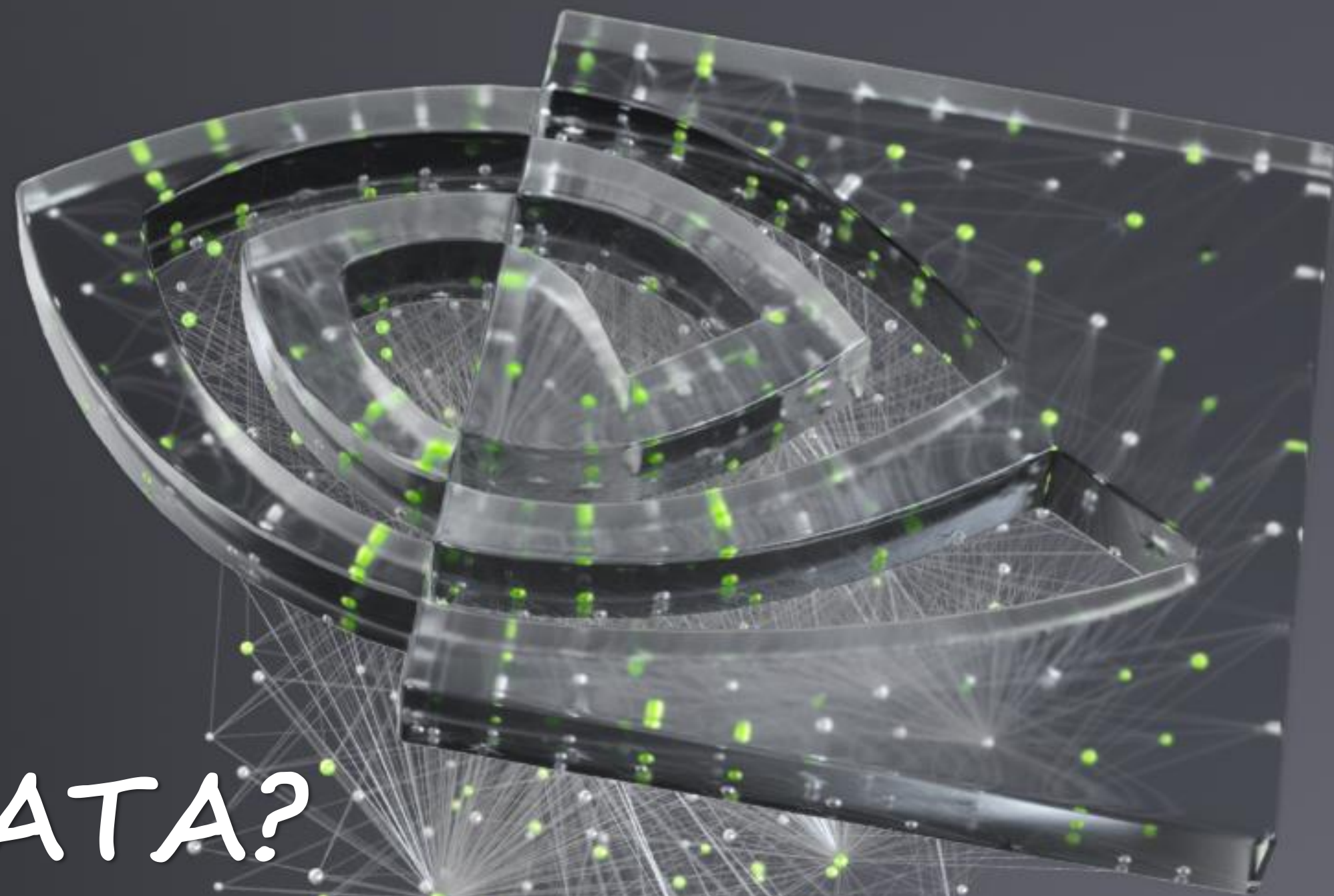




# WHY ~~HOW~~ GPU COMPUTING WORKS







WHERE'S MY DATA?

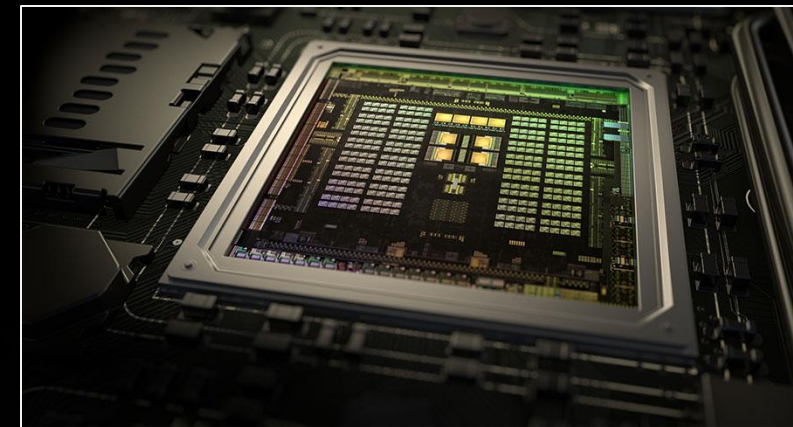
~~WHY~~  
HOW GPU COMPUTING WORKS

NOBODY CARES ABOUT FLOPs

REALLY  
ALMOST NOBODY CARES ABOUT FLOPs  
^

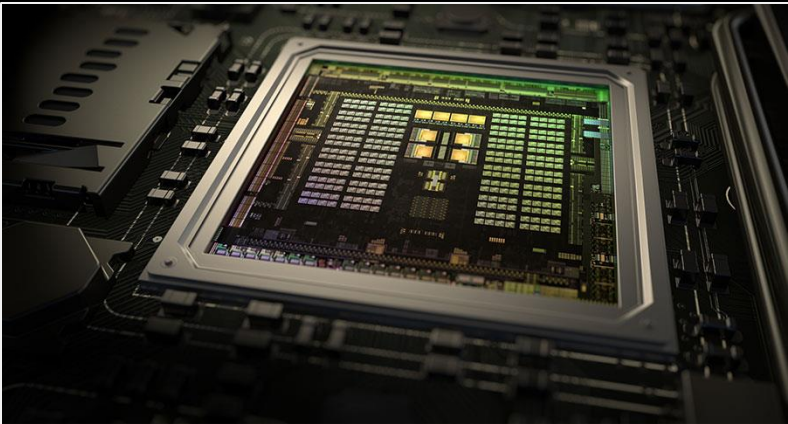


CPU

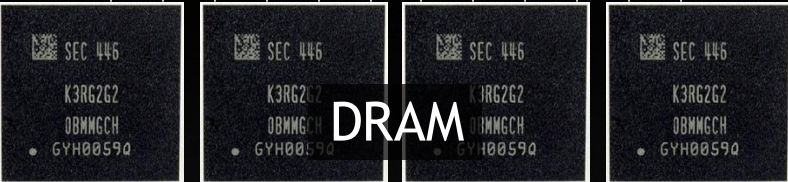


2000 GFLOPs FP64

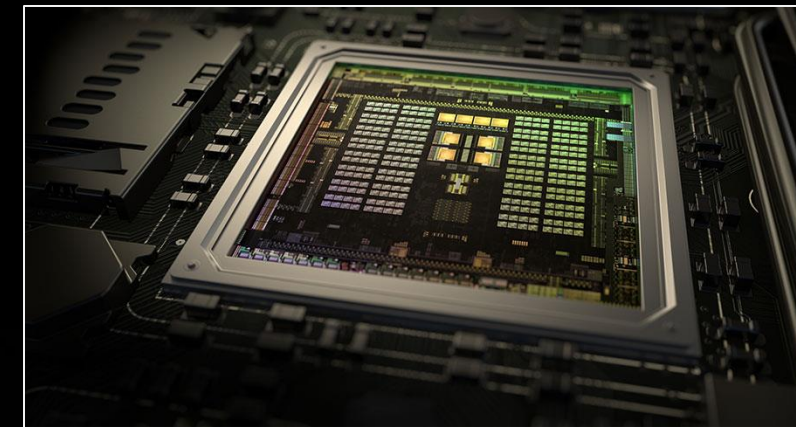
CPU



200 GBytes / sec



CPU



2000 GFLOPs FP64

200 GBytes / sec  
= 25 Giga-FP64 / sec  
(because FP64 = 8 bytes)





# THIS IS COMPUTE INTENSITY

How many operations must I do on some data to make it worth the cost of loading it?

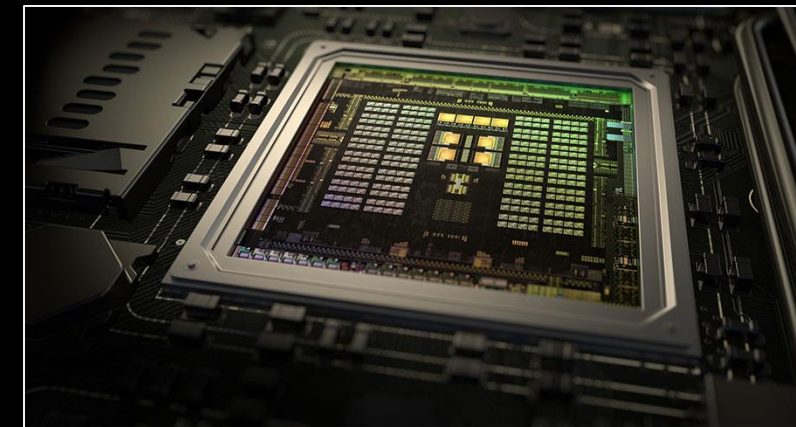
$$\text{Required Compute Intensity} = \frac{\text{FLOPs}}{\text{Data Rate}} = 80$$

2000 GFLOPs FP64



200 GBytes / sec  
= 25 Giga-FP64 / sec  
(because FP64 = 8 bytes)

CPU



DRAM

# THIS IS COMPUTE INTENSITY

How many operations must I do on some data to make it worth the cost of loading it?

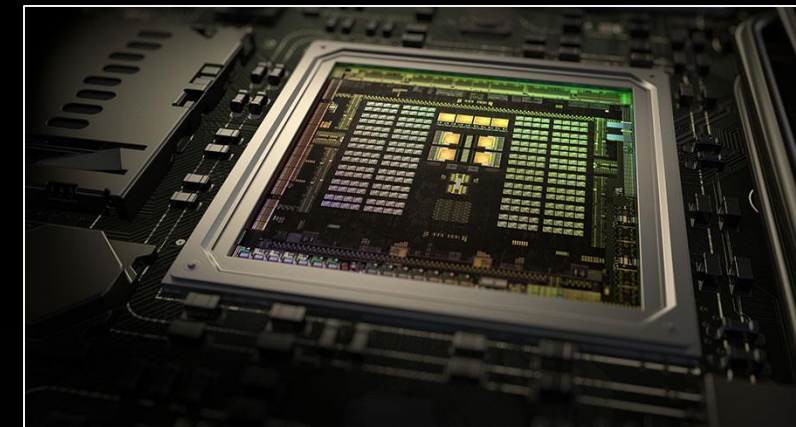
$$\text{Required Compute Intensity} = \frac{\text{FLOPs}}{\text{Data Rate}} = 80$$

2000 GFLOPs FP64



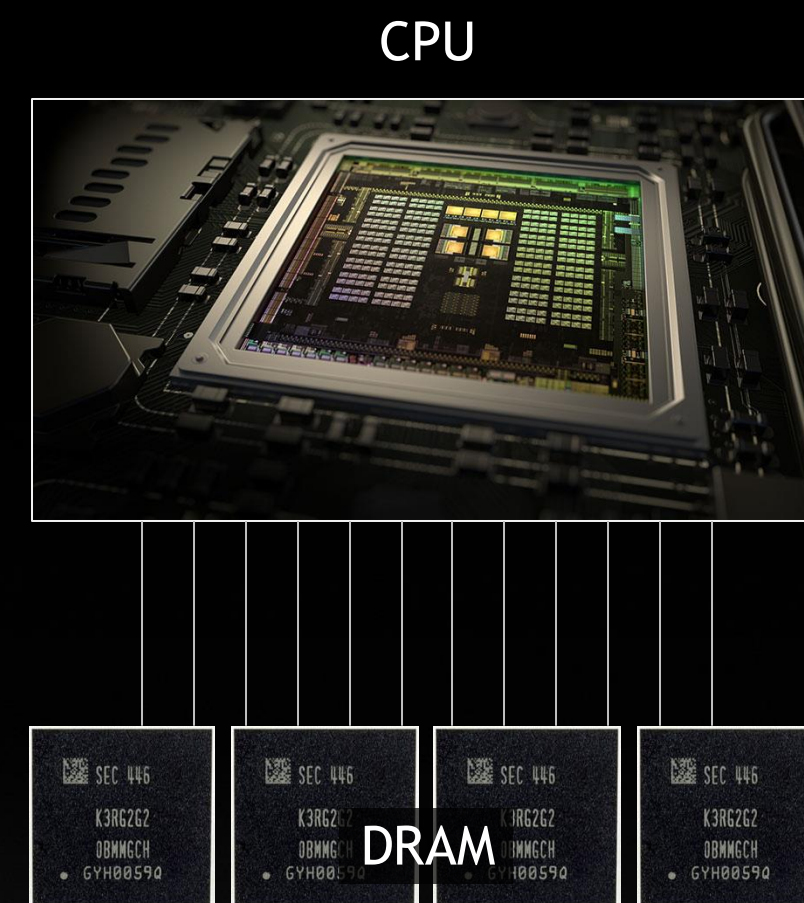
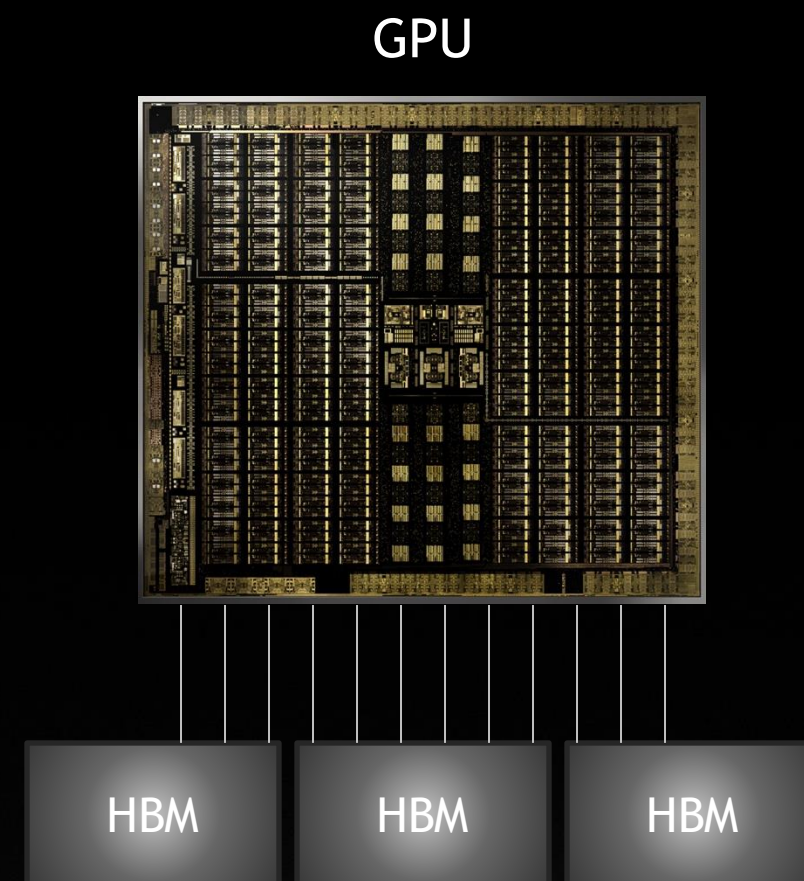
200 GBytes / sec  
= 25 Giga-FP64 / sec  
(because FP64 = 8 bytes)

CPU



So for every number I load from memory, I need to do 80 operations on it to break even





	NVIDIA A100	Intel Xeon 8280	AMD Rome 7742
Peak FP64 GigaFLOPs	19500	2190	2300
Memory B/W (GB/sec)	1555	131	204
Compute Intensity	100	134	90



REALLY  
ALMOST NOBODY CARES ABOUT FLOPs  
^

...BECAUSE WE SHOULD REALLY BE CARING ABOUT  
MEMORY BANDWIDTH

REALLY  
ALMOST NOBODY CARES ABOUT FLOPs  
^

...BECAUSE WE SHOULD REALLY BE CARING ABOUT  
~~MEMORY BANDWIDTH~~ LATENCY

# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)



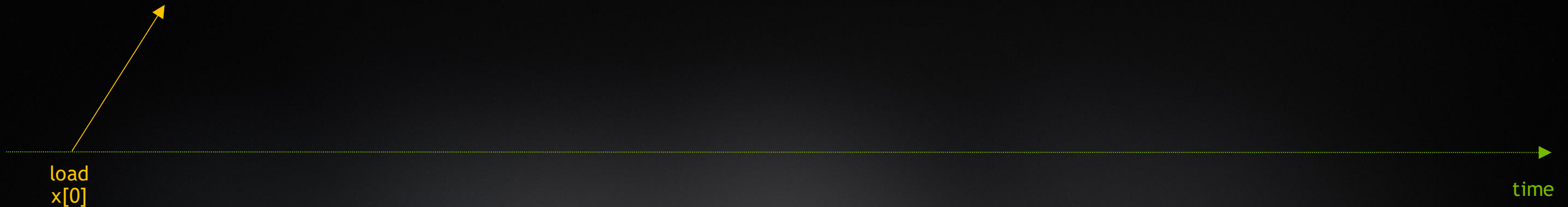
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)



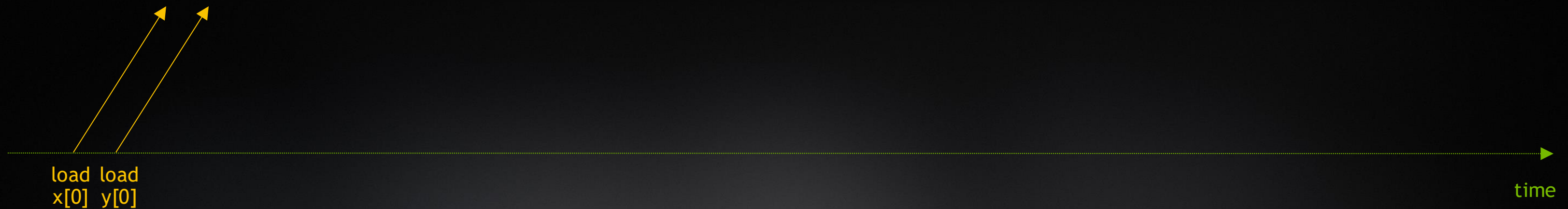
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)



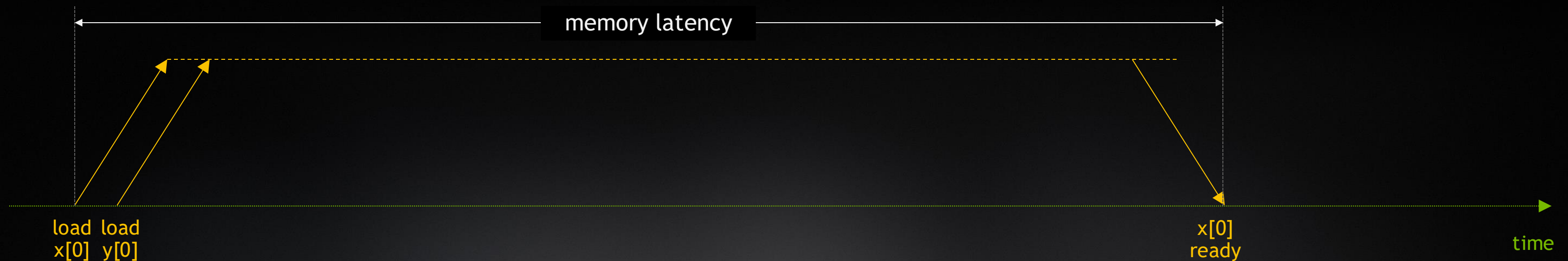
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)





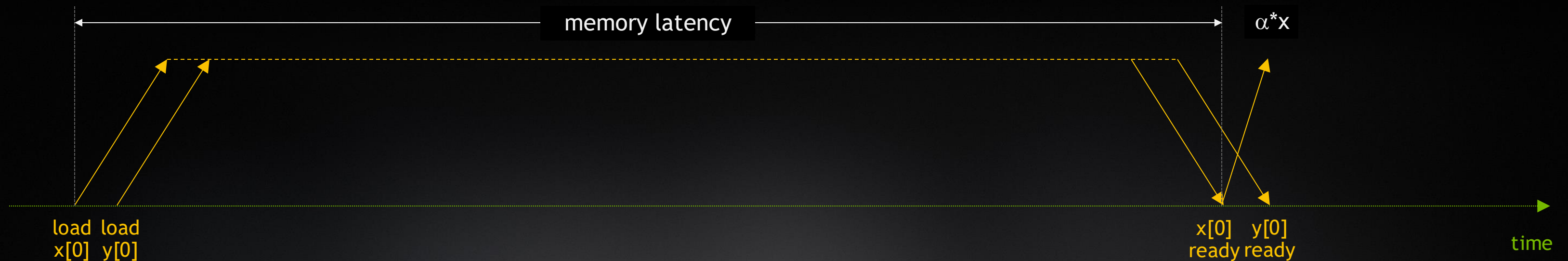
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)



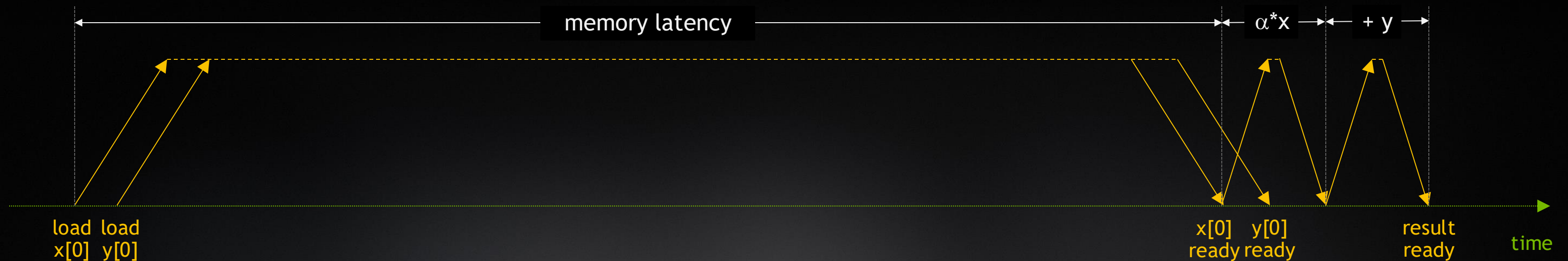
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

2 FLOPs: multiply & add

2 memory loads: x[i] & y[i] (per element)

Single operation: FMA (fused multiply-add)



*SPEED OF LIGHT = 300,000,000 M/S*



SPEED OF LIGHT = 300,000,000 M/S  
COMPUTER CLOCK = 3,000,000,000 Hz

*SPEED OF LIGHT = 300,000,000 M/S*  
*COMPUTER CLOCK = 3,000,000,000 Hz*

So in 1 clock tick light travels 100mm (~4 inches)

SPEED OF LIGHT = 300,000,000 M/S

COMPUTER CLOCK = 3,000,000,000 Hz

SPEED OF ELECTRICITY = 60,000,000 M/S  
IN SILICON

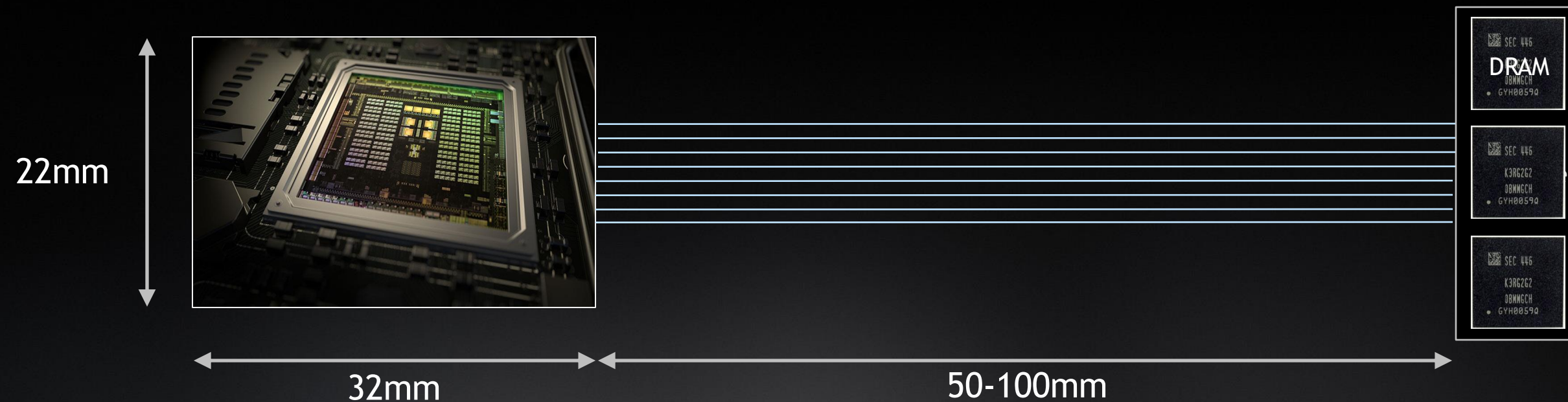
So in 1 clock tick electricity travels 20mm (~0.8 inches)



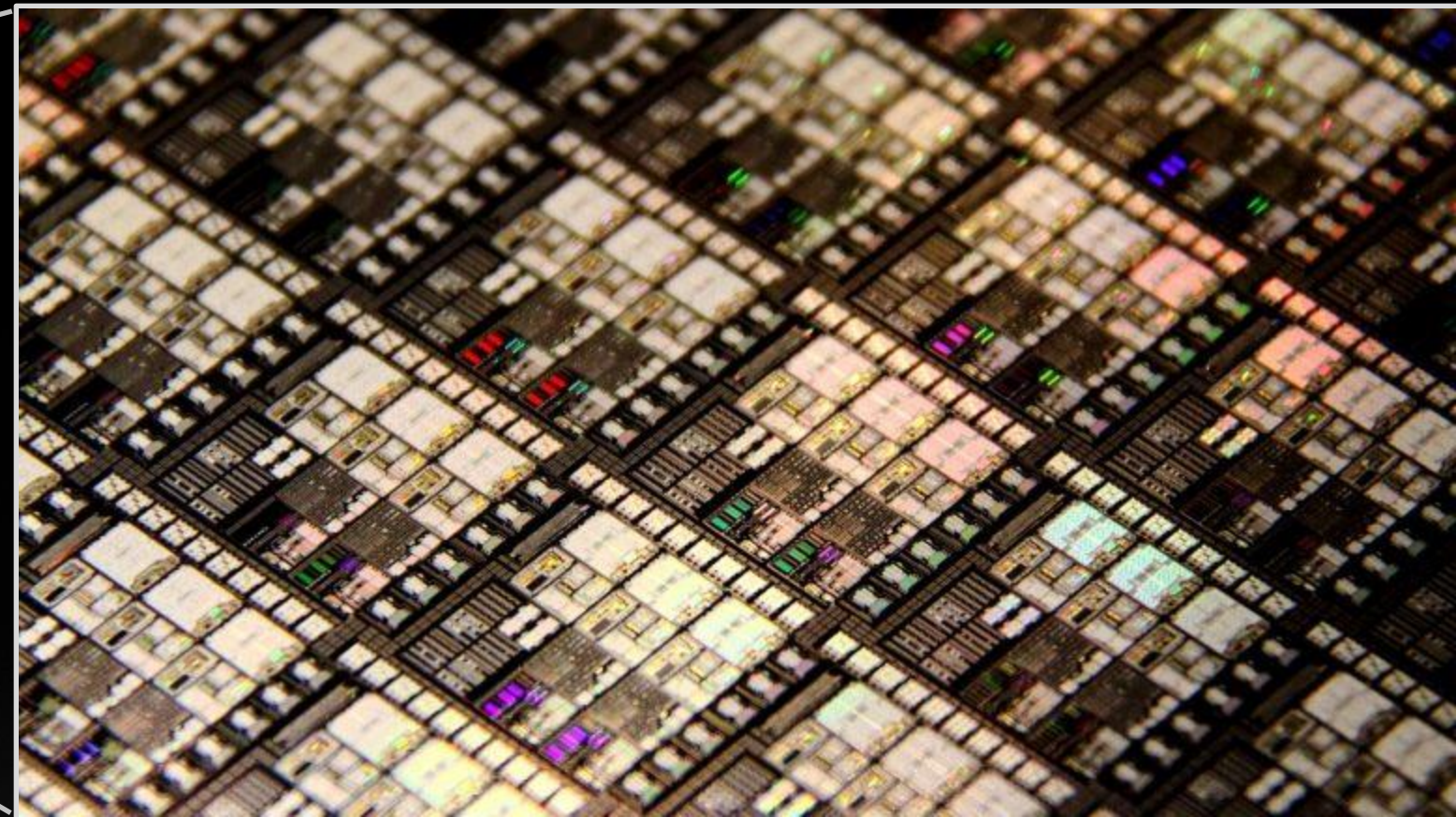
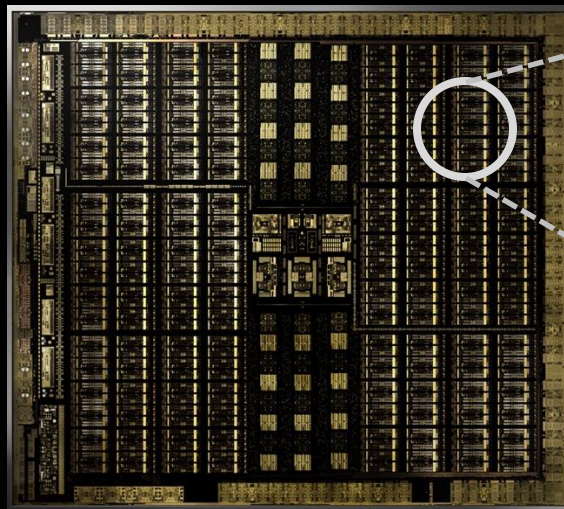
SPEED OF LIGHT = 300,000,000 M/S

COMPUTER CLOCK = 3,000,000,000 Hz

SPEED OF ELECTRICITY = 60,000,000 M/S  
IN SILICON



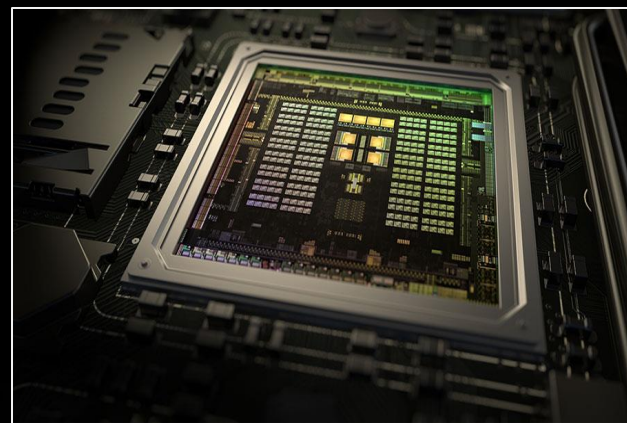






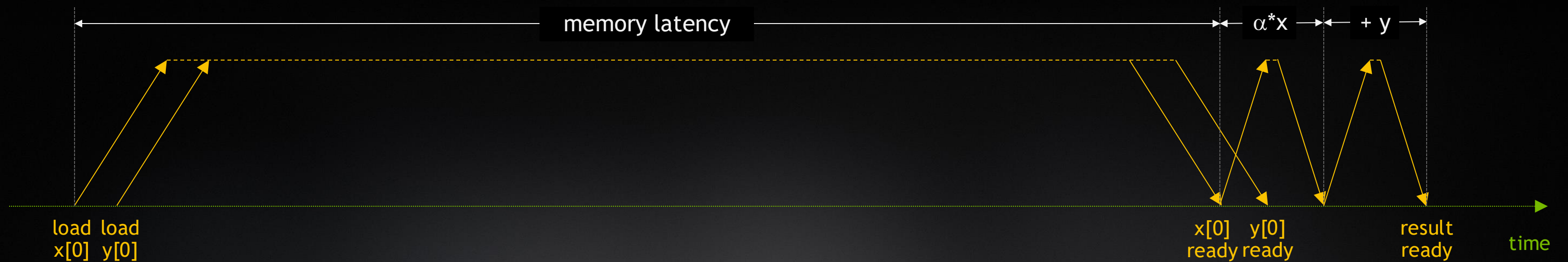
# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

Intel Xeon 8280



Memory bandwidth: 131 GB/sec

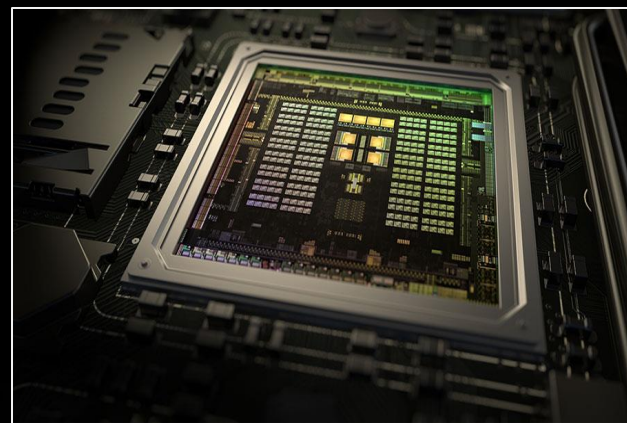
Memory latency: 89 ns





# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

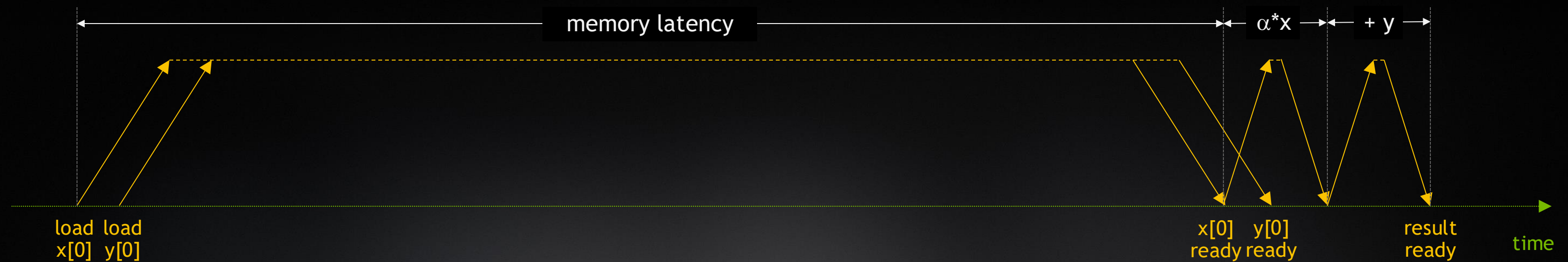
Intel Xeon 8280



Memory bandwidth: 131 GB/sec

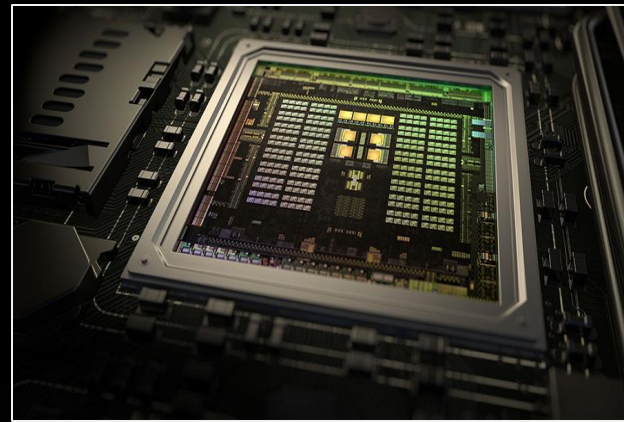
Memory latency: 89 ns

11,659 bytes can be moved in 89ns



# DAXPY: $\alpha \underline{X} + \underline{Y} = \underline{Z}$

Intel Xeon 8280



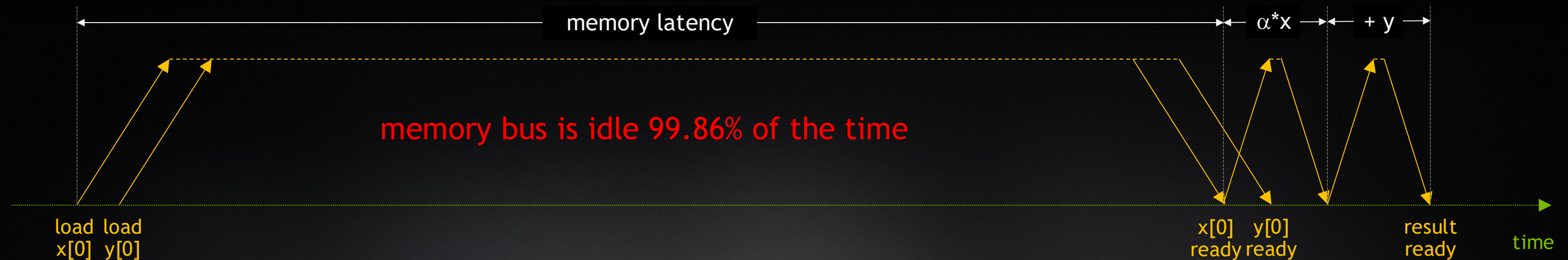
Memory bandwidth: 131 GB/sec

Memory latency: 89 ns

11,659 bytes can be moved in 89ns

*daxpy* moves 16 bytes per 89ns latency

Memory efficiency = 0.14%



# COMPARISON OF DAXPY\* EFFICIENCY ON DIFFERENT CHIPS

	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	131
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	11,659
Memory efficiency	0.0025%	0.064%	0.14%

*\*daxpy moves 16 bytes per latency*



# SO WHAT CAN WE DO ABOUT IT?

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

11,659 bytes can be moved in 89ns

*daxpy* moves 16 bytes per 89ns latency

Memory efficiency = 0.14%

To keep memory bus busy, we must run  $\frac{11,659}{16} = 729$  iterations at once

# LOOP UNROLLING

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i += 8 )
    {
        y[i+0] = alpha * x[i+0] + y[i+0];
        y[i+1] = alpha * x[i+1] + y[i+1];
        y[i+2] = alpha * x[i+2] + y[i+2];
        y[i+3] = alpha * x[i+3] + y[i+3];
        y[i+4] = alpha * x[i+4] + y[i+4];
        y[i+5] = alpha * x[i+5] + y[i+5];
        y[i+6] = alpha * x[i+6] + y[i+6];
        y[i+7] = alpha * x[i+7] + y[i+7];
    }
}
```

Compilers rarely unroll a loop **729** times

Just one thread issuing all these commands

One thread cannot hold **729** outstanding loads

To keep memory bus busy, we must run  $\frac{11,659}{16} = \mathbf{729}$  iterations at once

# THE ONLY OPTION IS THREADS

```
void daxpy(int n, double alpha, double *x, double *y)
{
    parallel for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

Each thread issues load operations independently

Ideally requires **729** threads

Limited by max threads & memory requests

To keep memory bus busy, we must run  $\frac{11,659}{16} = \mathbf{729}$  iterations at once

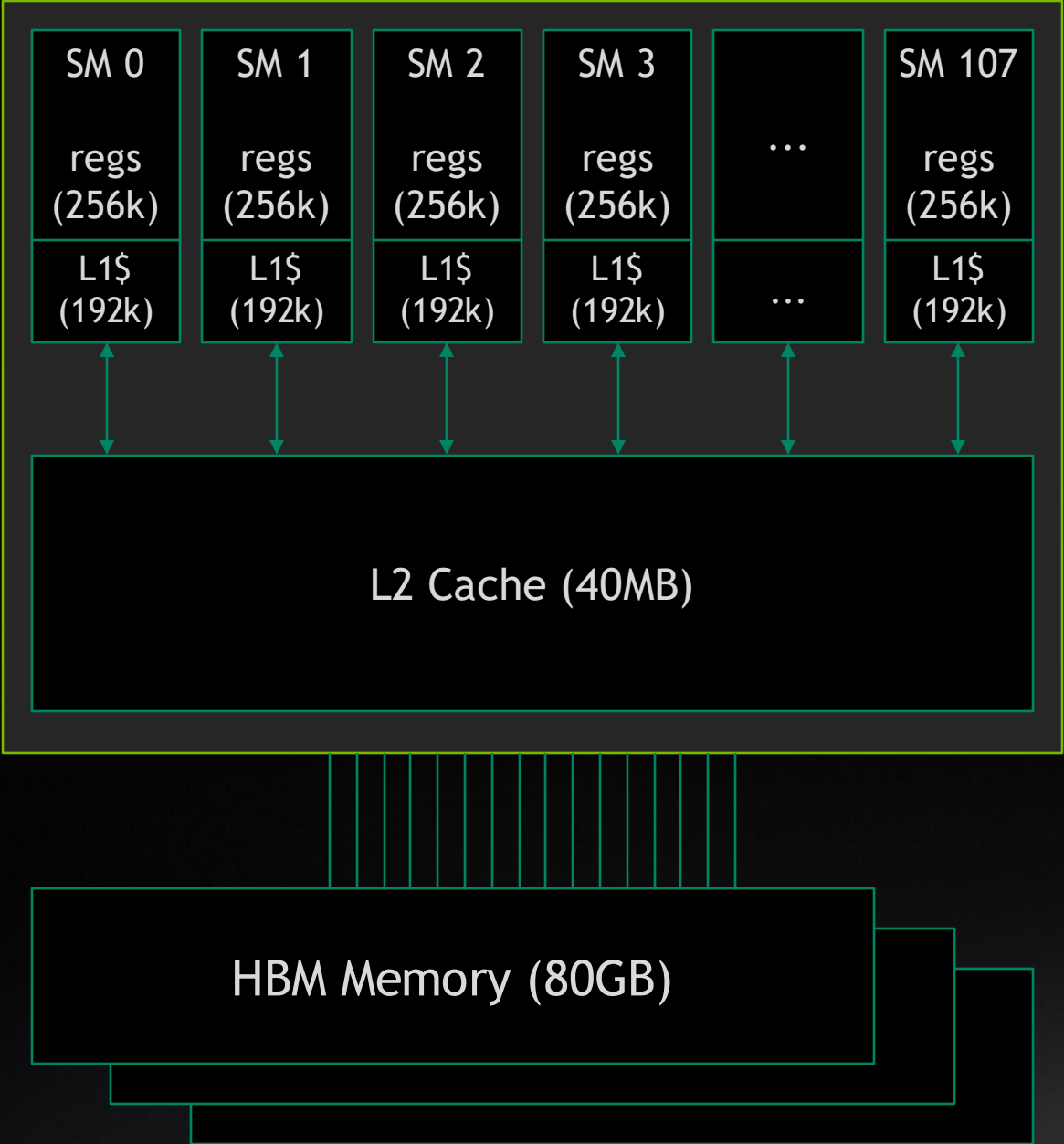


# COMPARISON OF DAXPY\* EFFICIENCY ON DIFFERENT CHIPS

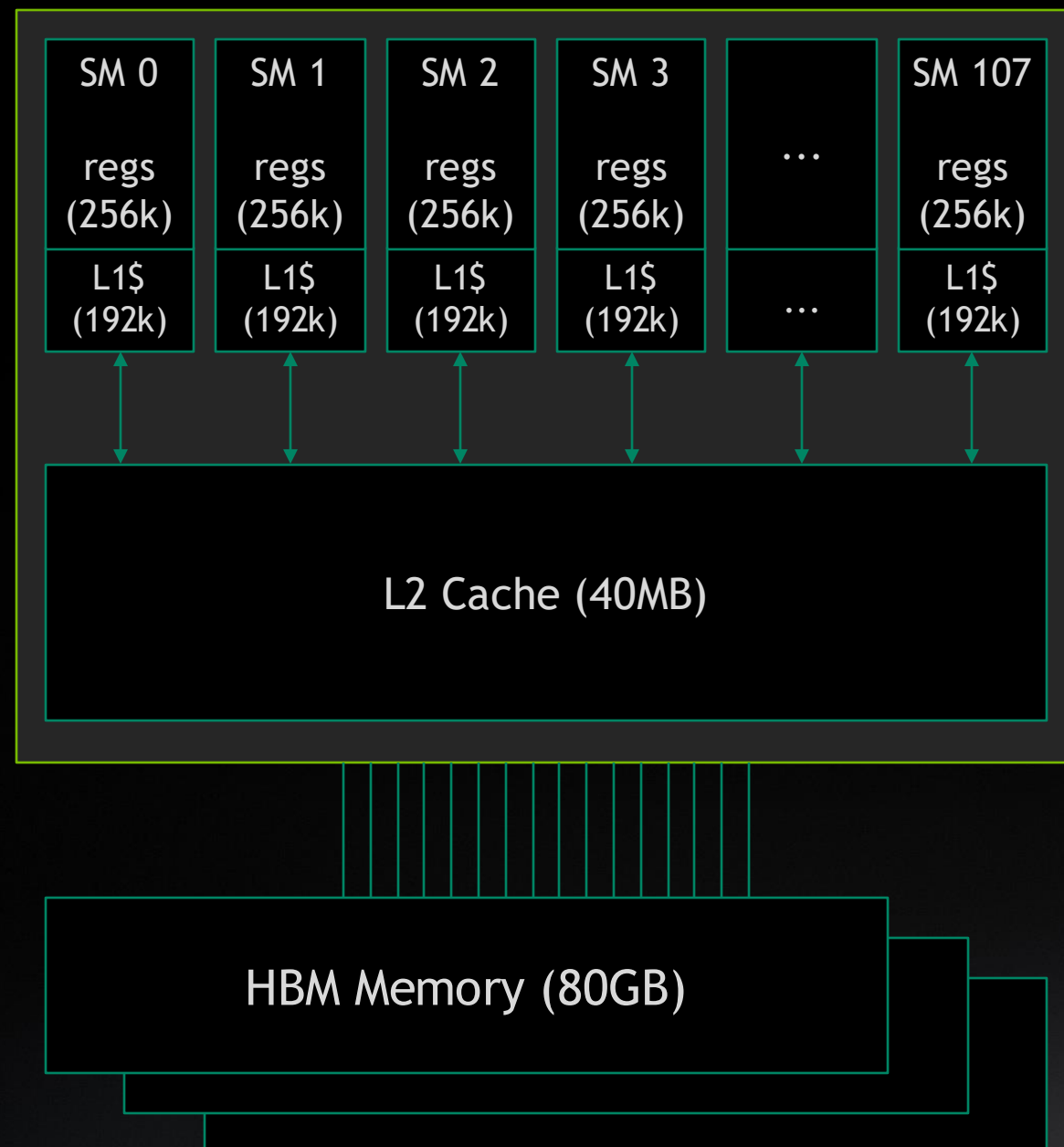
	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x

*\*daxpy moves 16 bytes per latency*

Ampere A100 GPU



## Ampere A100 GPU



108 Streaming Multiprocessors (SMs)

256kB Register File per SM (27MB total)

192kB L1 Cache & Shared Memory per SM  
(20MB total)

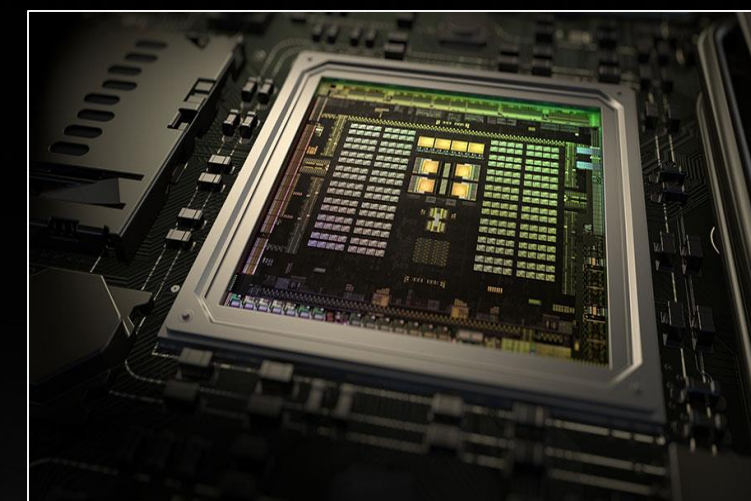
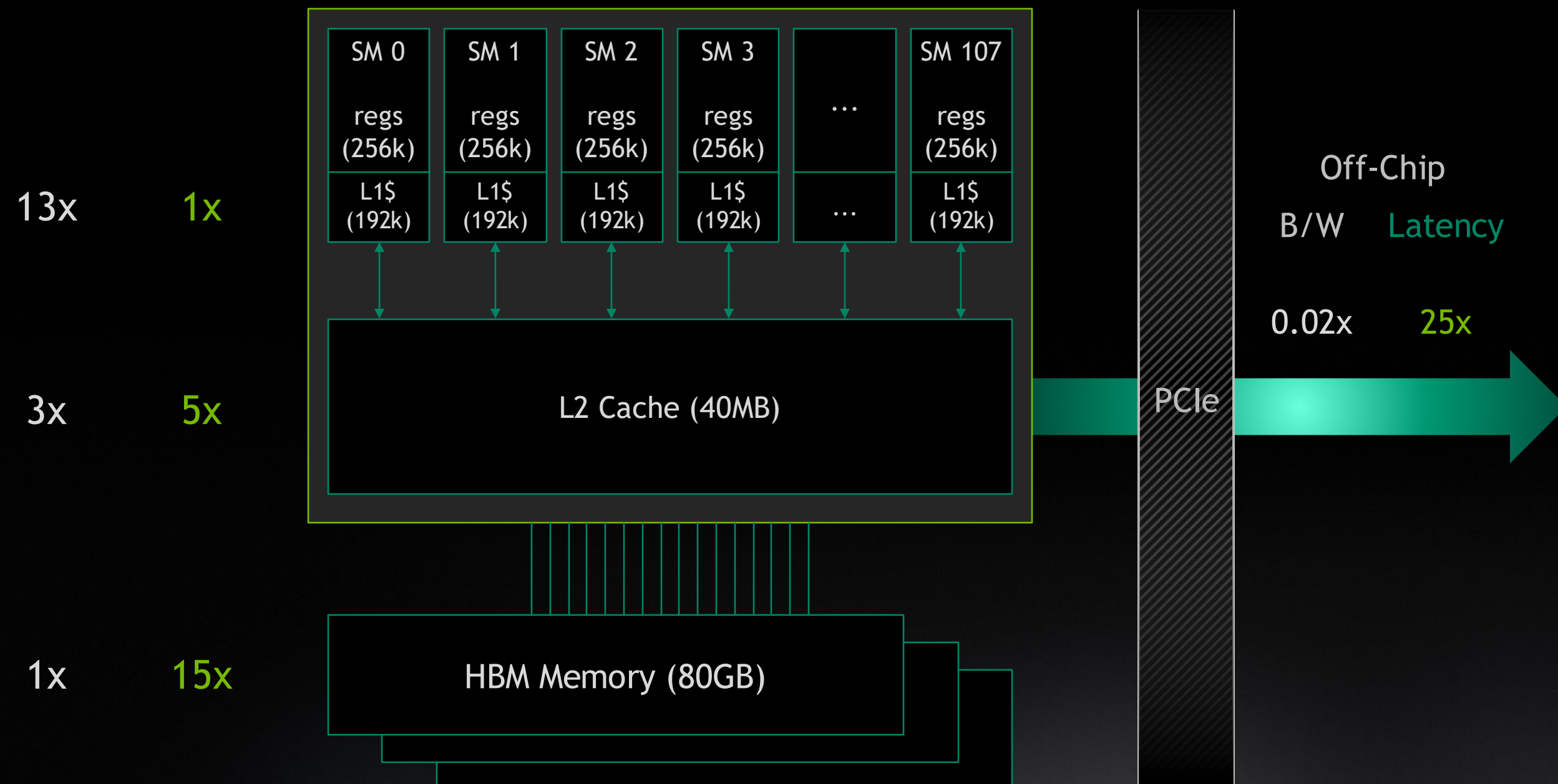
40MB L2 cache shared across all SMs

caches

80GB High Bandwidth Memory

B/W    Latency

## Ampere A100 GPU



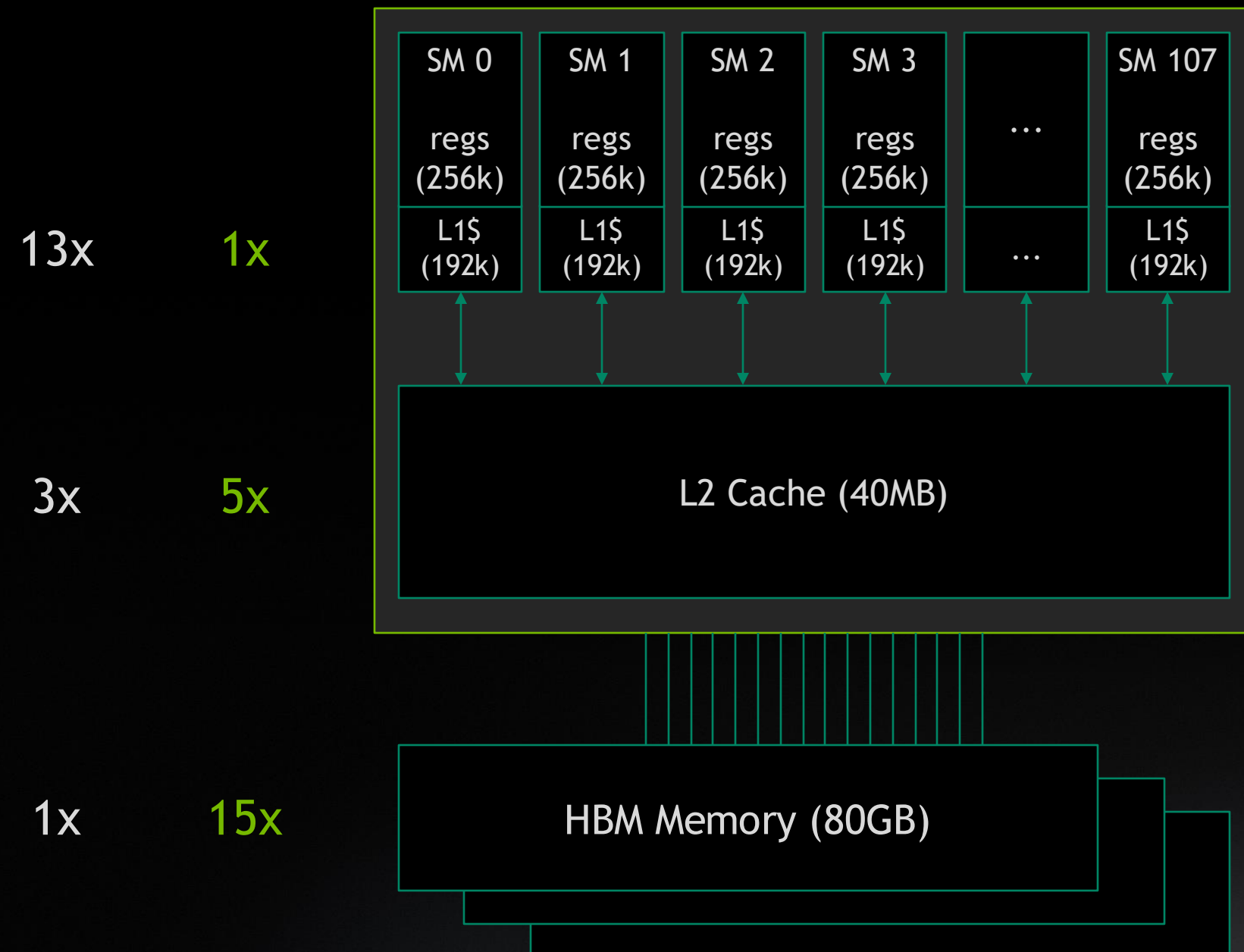
1x    15x

HBM Memory (80GB)



B/W Latency

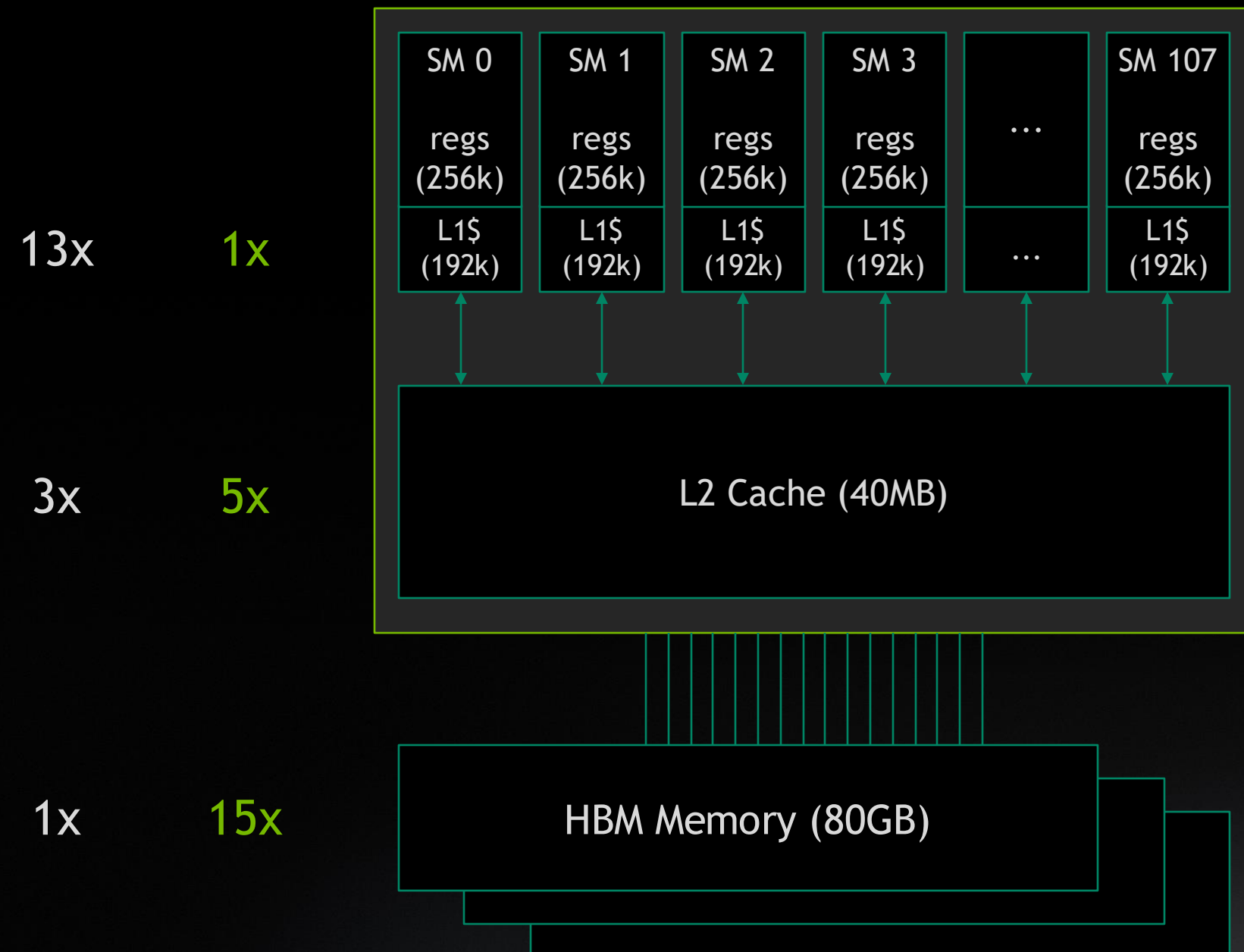
## Ampere A100 GPU



Data Location	Bandwidth (GB/sec)	Compute Intensity
L1 Cache	19,400	8
L2 Cache	4,000	39
HBM	1,555	100
NVLink	300	520
PCIe	25	6240

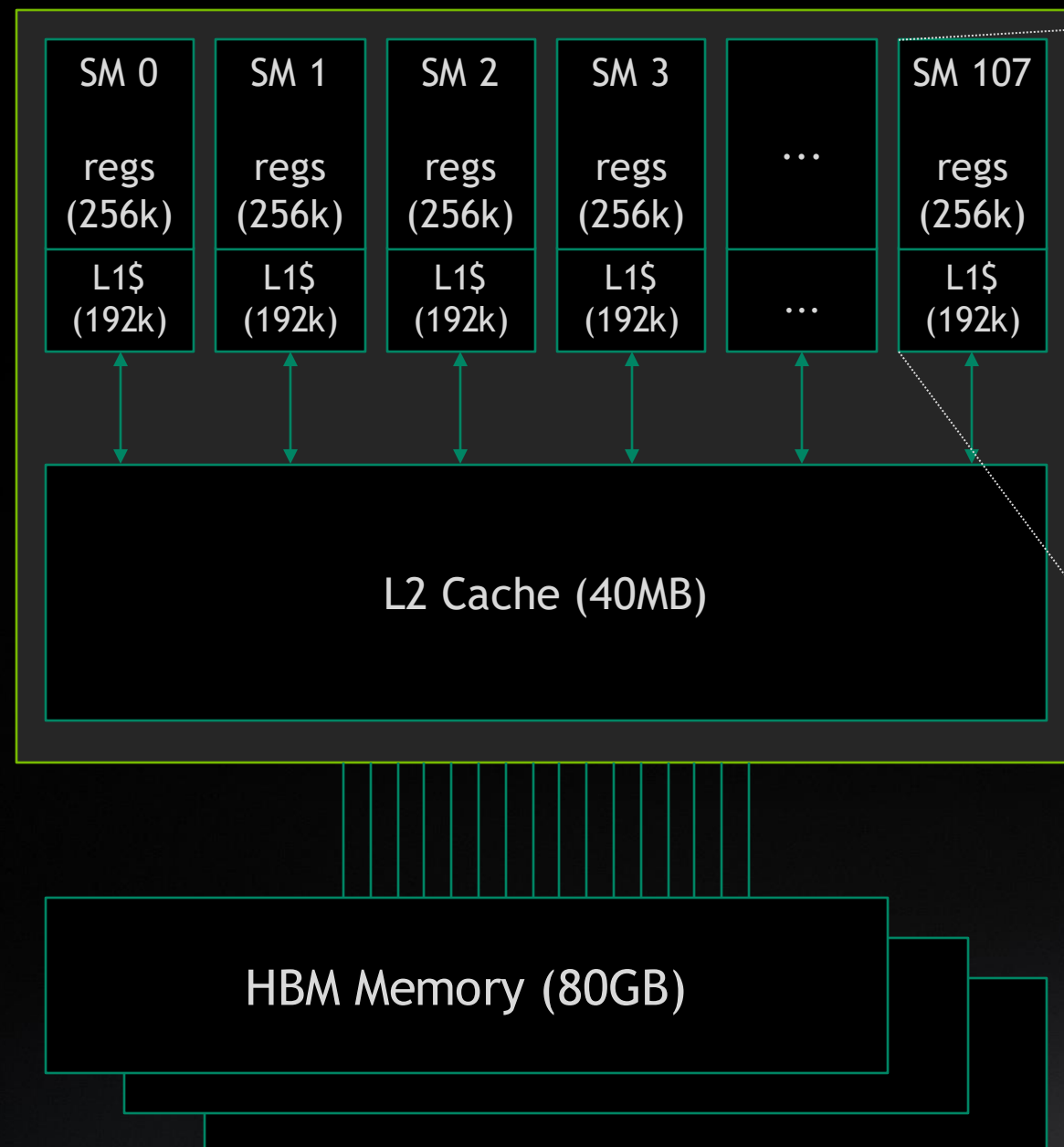
B/W Latency

## Ampere A100 GPU



Data Location	Latency (ns)	Threads Required
L1 Cache	27	32,738
L2 Cache	150	37,500
HBM	404	39,264
NVLink	700	13,125
PCIe	1470	2297

## Ampere A100 GPU



## A100 Streaming Multiprocessor (SM)



64 warps/SM

4x concurrent warp exec

64k x 4-byte registers

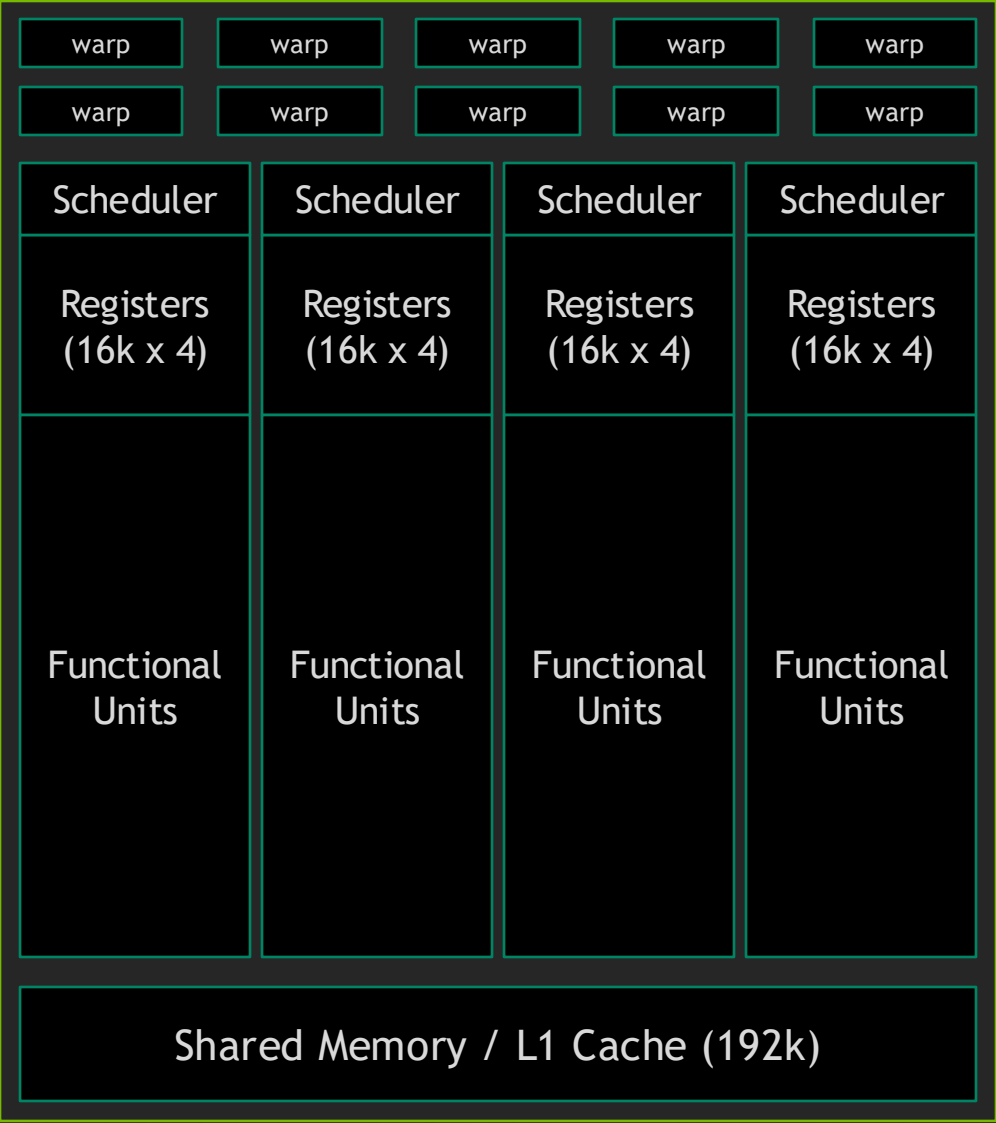
192KB L1/shared memory  
(configurable split)

The GPU runs threads in groups of **32** - each group is known as a **warp**

# THE GPU'S SECRET SAUCE: OVERSUBSCRIPTION

	Per SM	On A100
Total Threads	2048	221,184
Total Warps	64	6,912
Active Warps	4	432
Waiting Warps	60	6,480
Active Threads	128	13,824
Waiting Threads	1,920	207,360

A100 Streaming Multiprocessor (SM)



64 warps/SM

4x concurrent warp exec

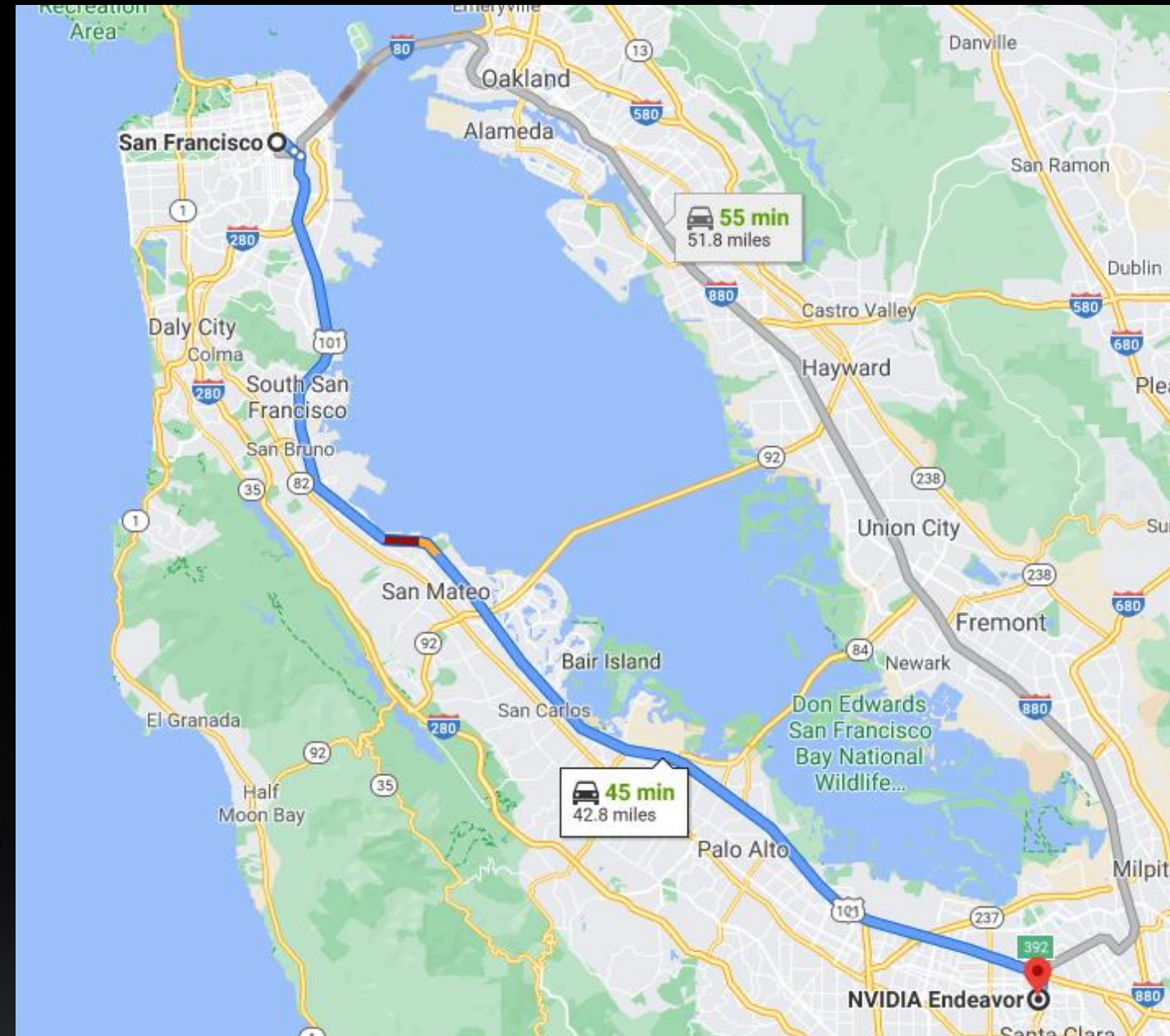
64k x 4-byte registers

192KB L1/shared memory  
(configurable split)

The GPU can switch from one warp to the next in a **single clock cycle**



# THROUGHPUT VS. LATENCY







San Francisco

Millbrae

Burlingame

Belmont

Redwood City

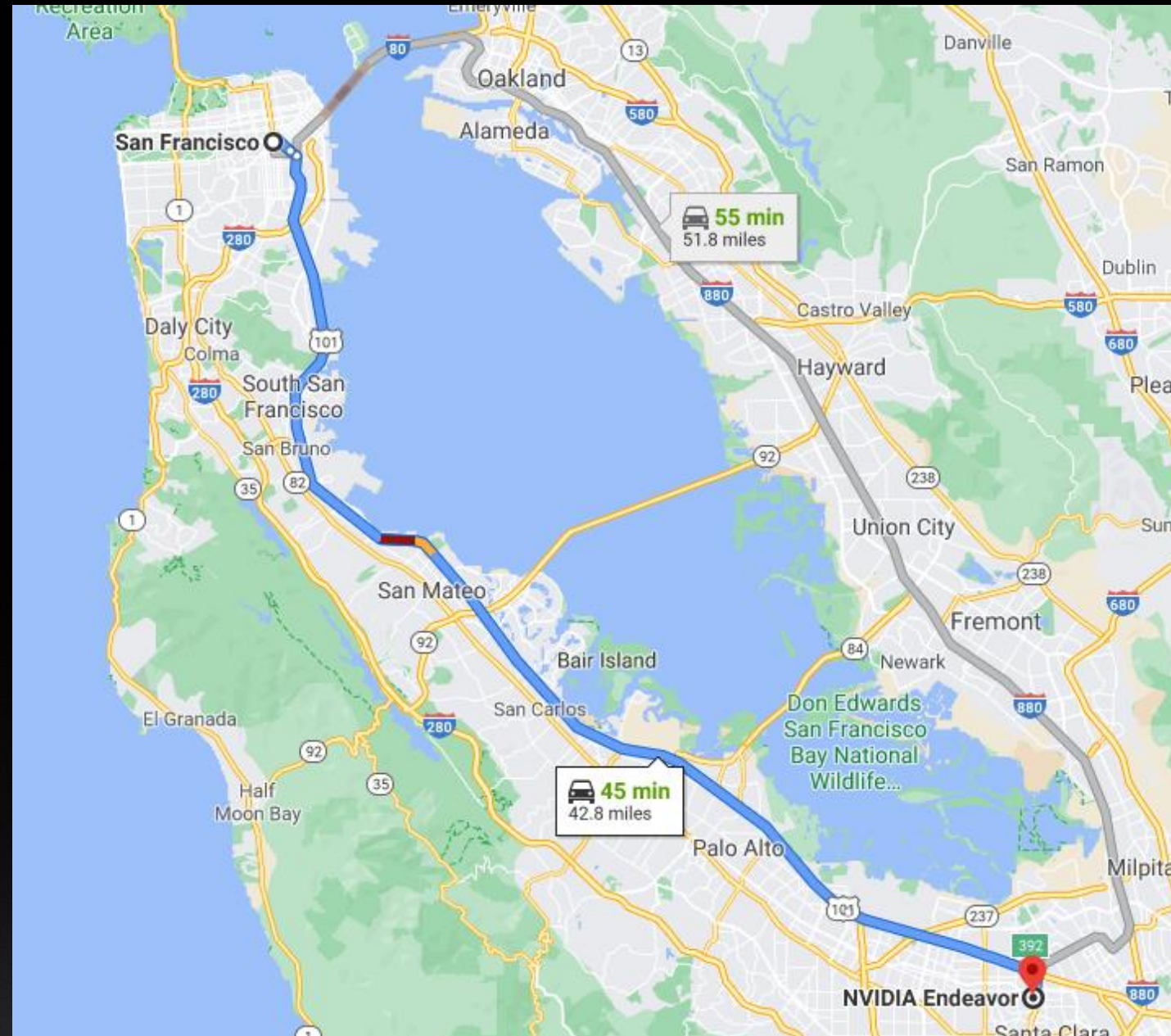
Palo Alto

Mountain View

Sunnyvale

NVIDIA

73 Minutes



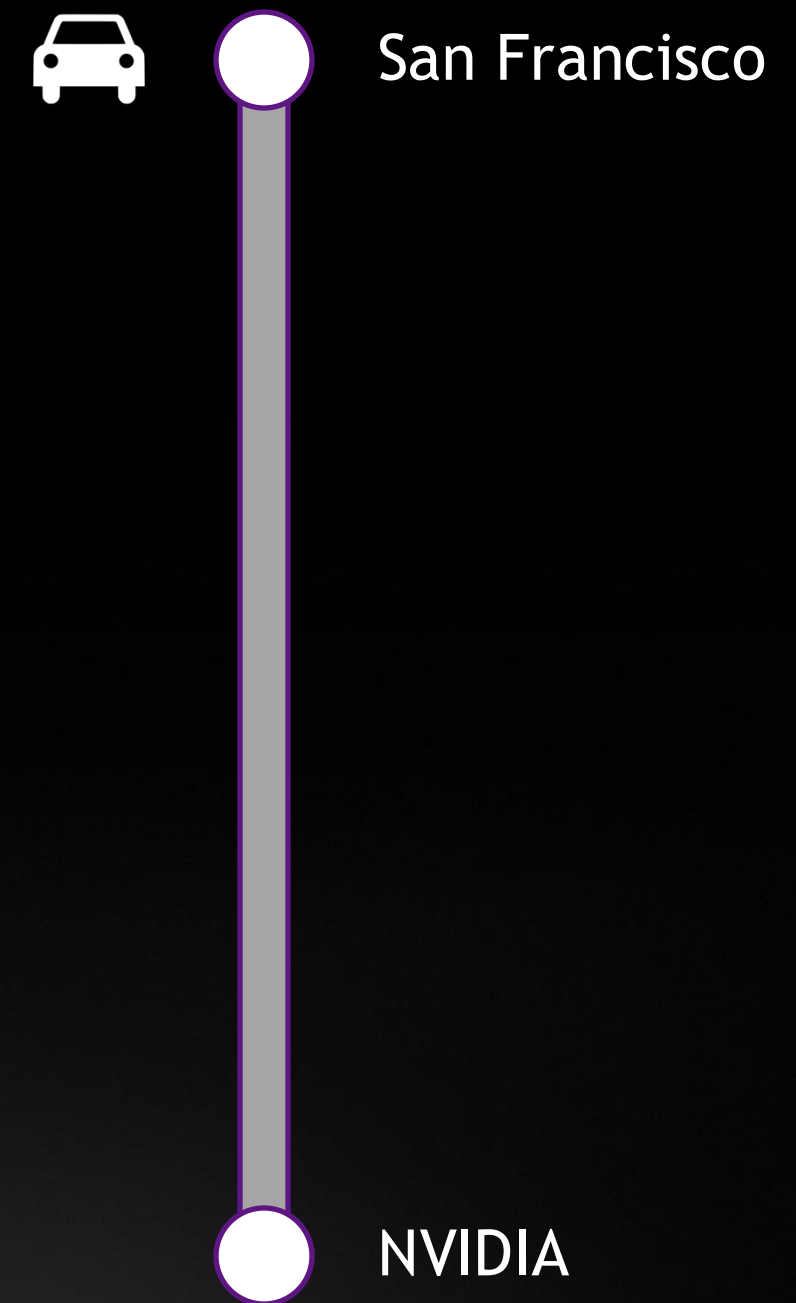
San Francisco

NVIDIA

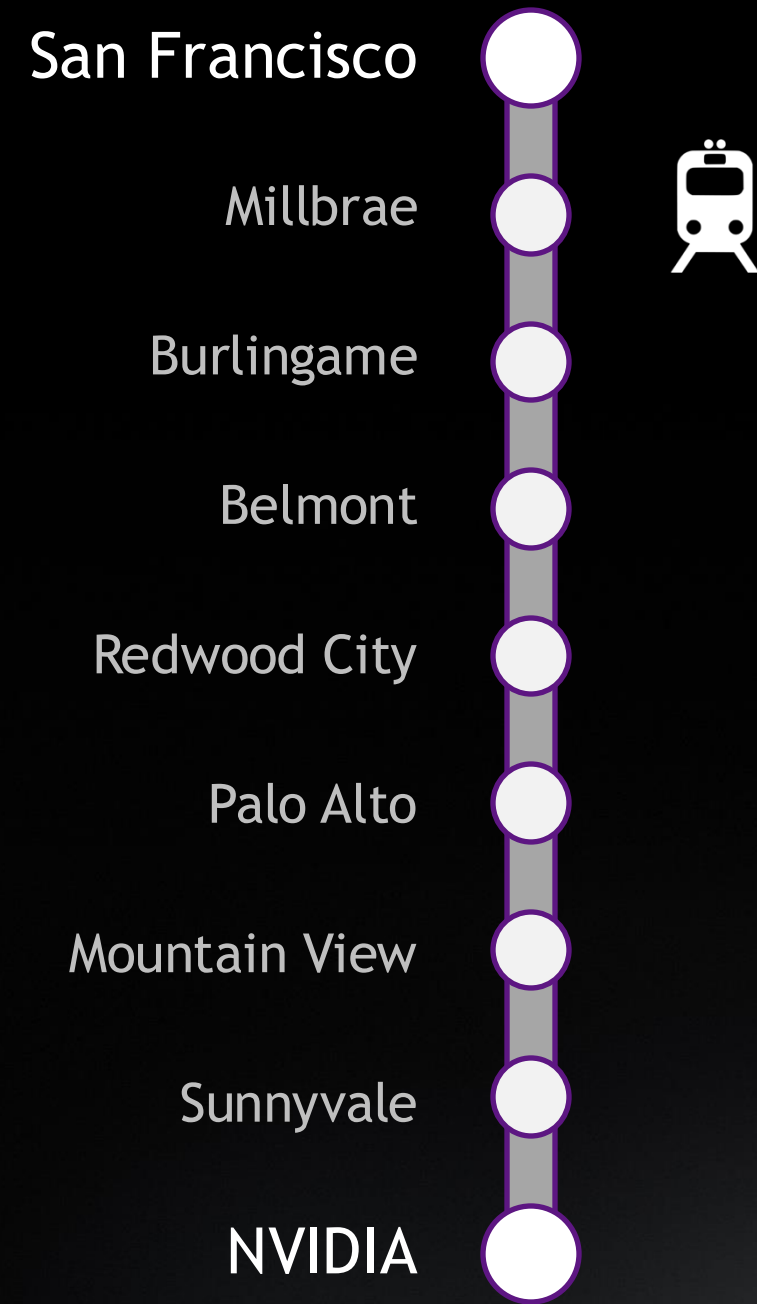
45 Minutes



73 Minutes



45 Minutes



73 Minutes



45 Minutes

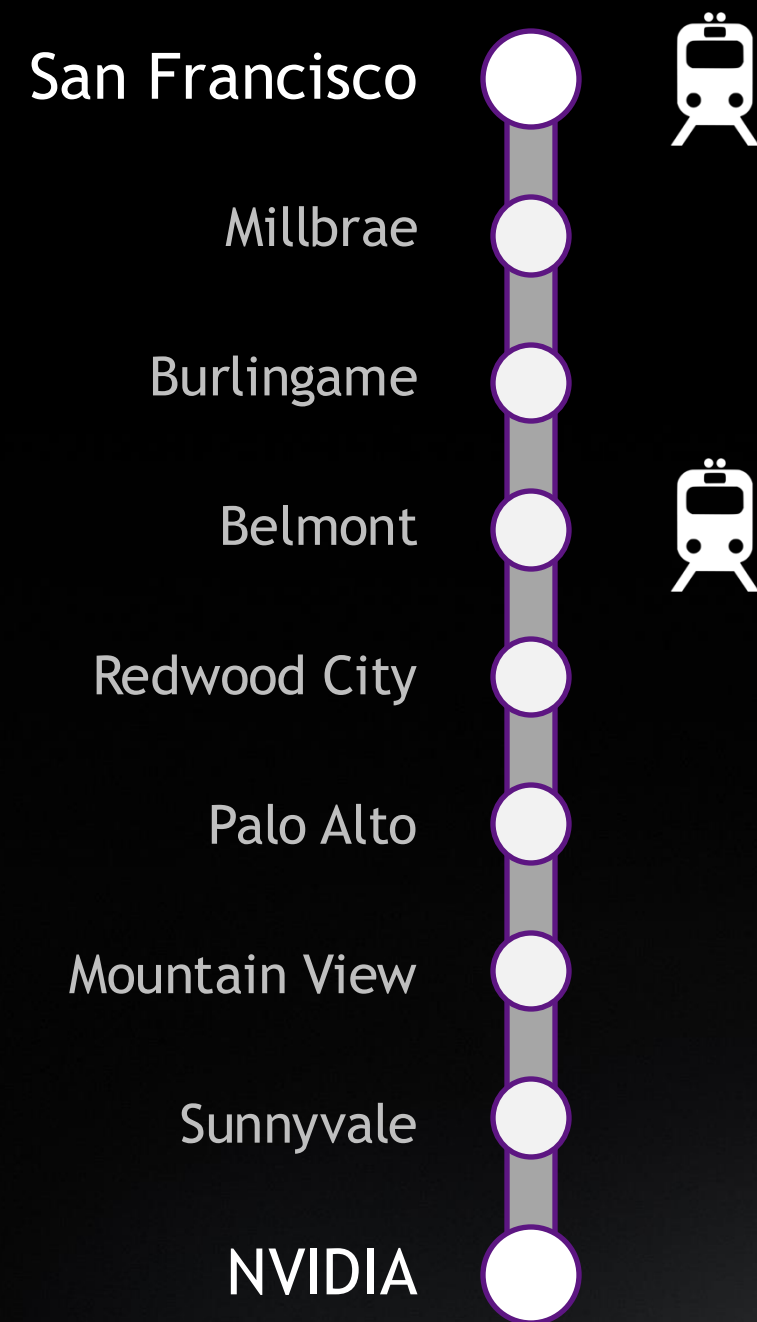




73 Minutes



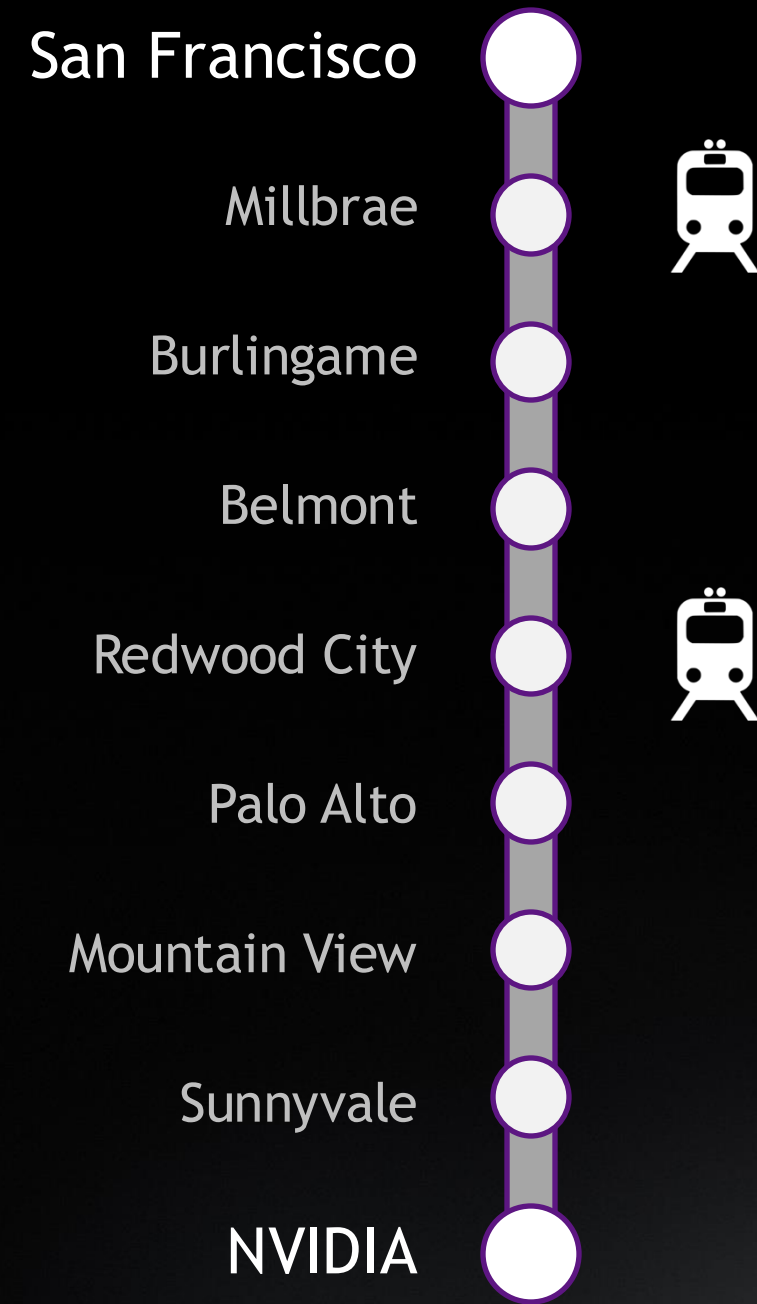
45 Minutes



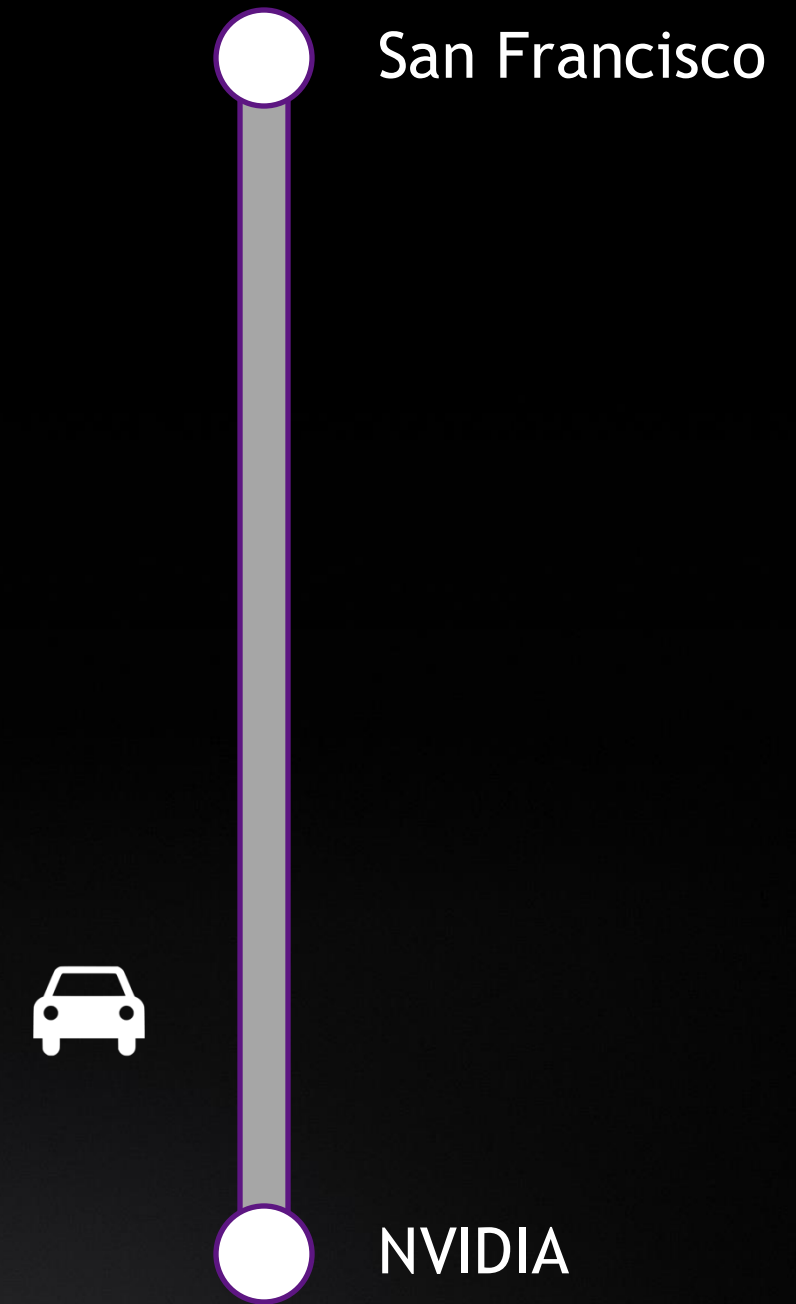
73 Minutes



45 Minutes

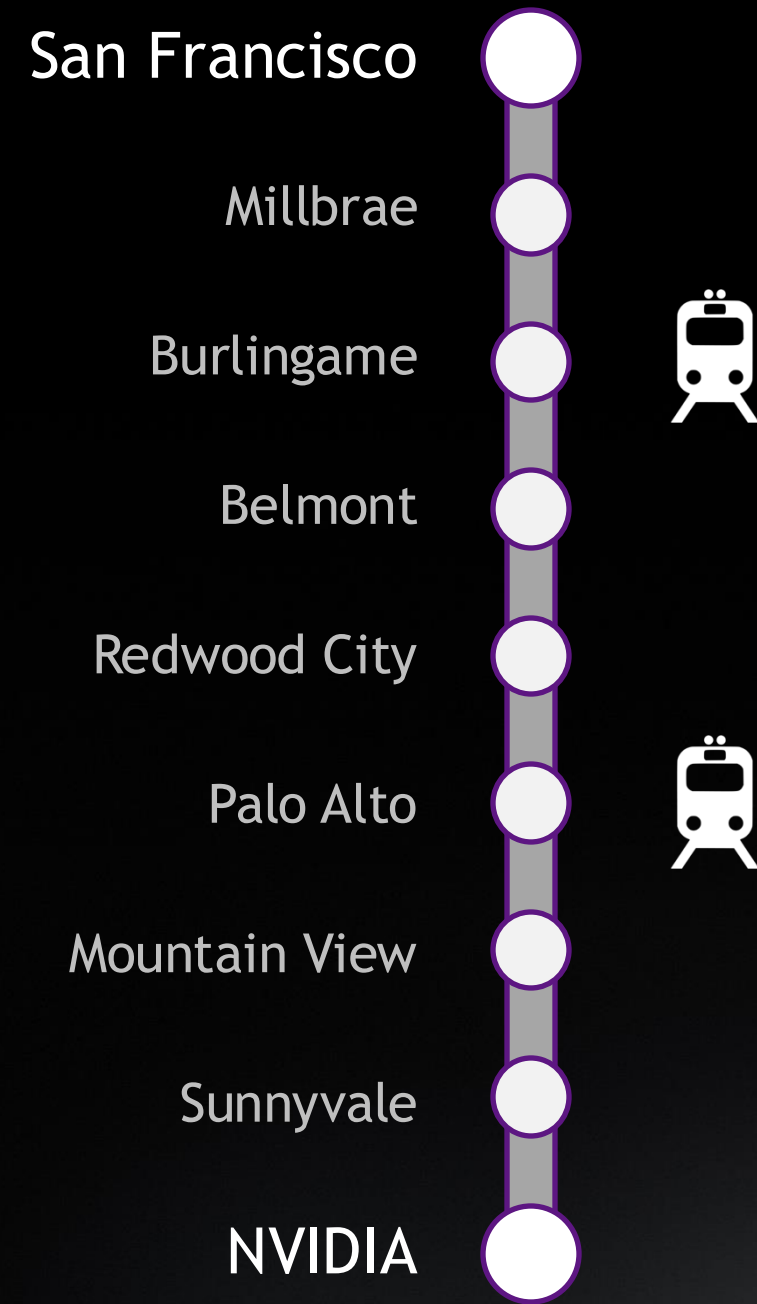


73 Minutes

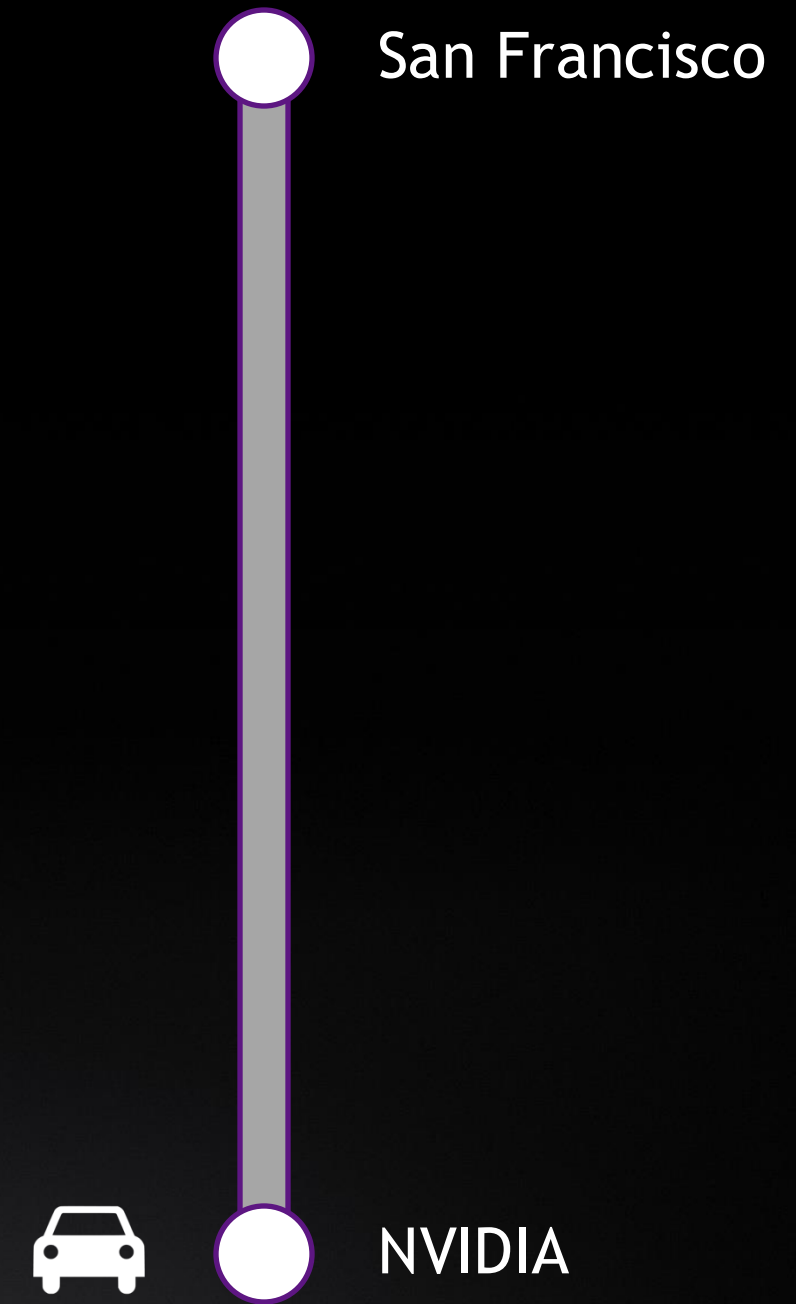


45 Minutes





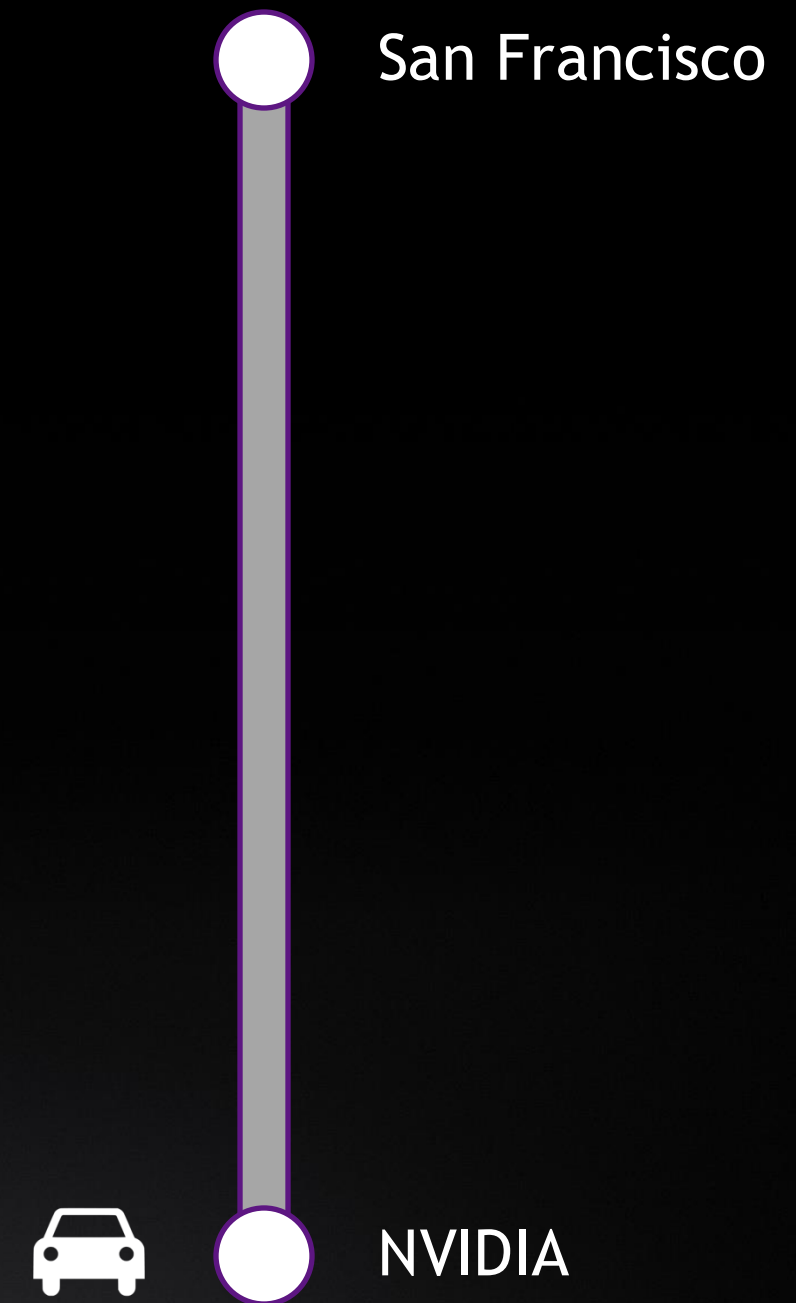
73 Minutes



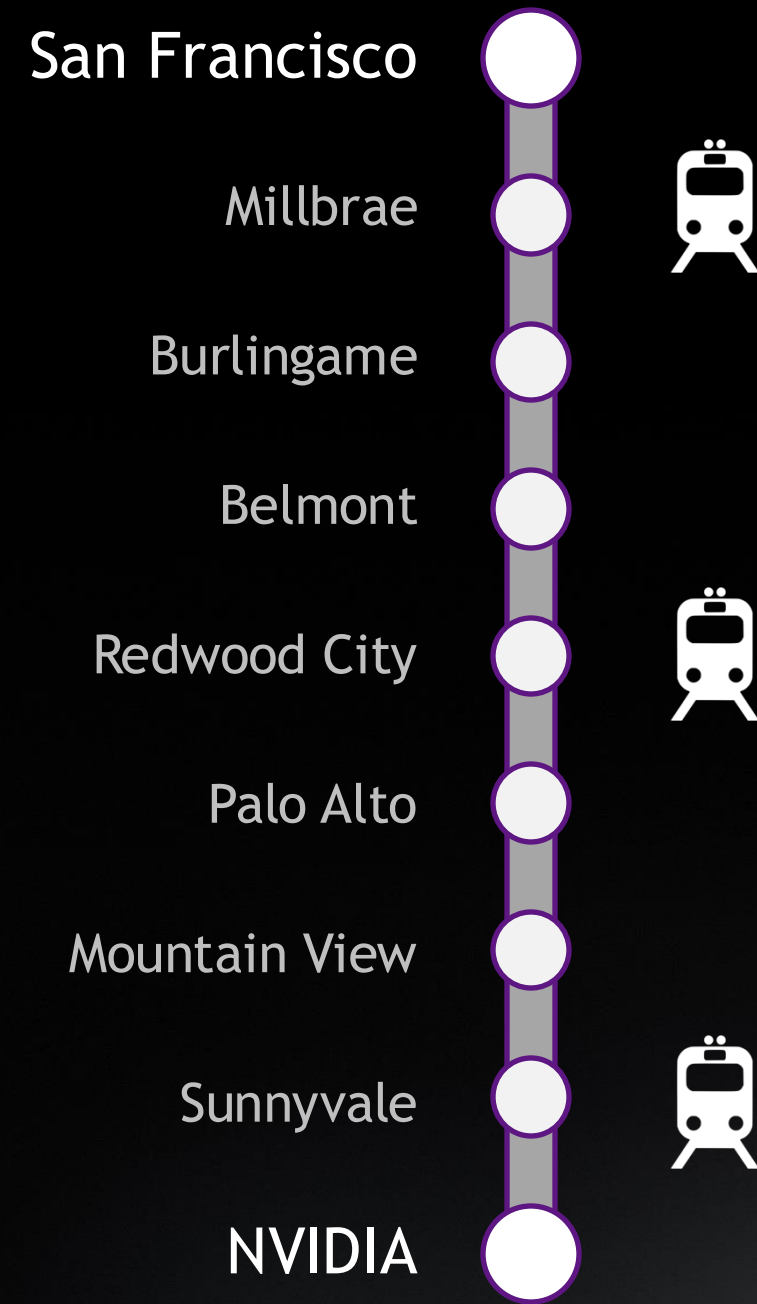
45 Minutes



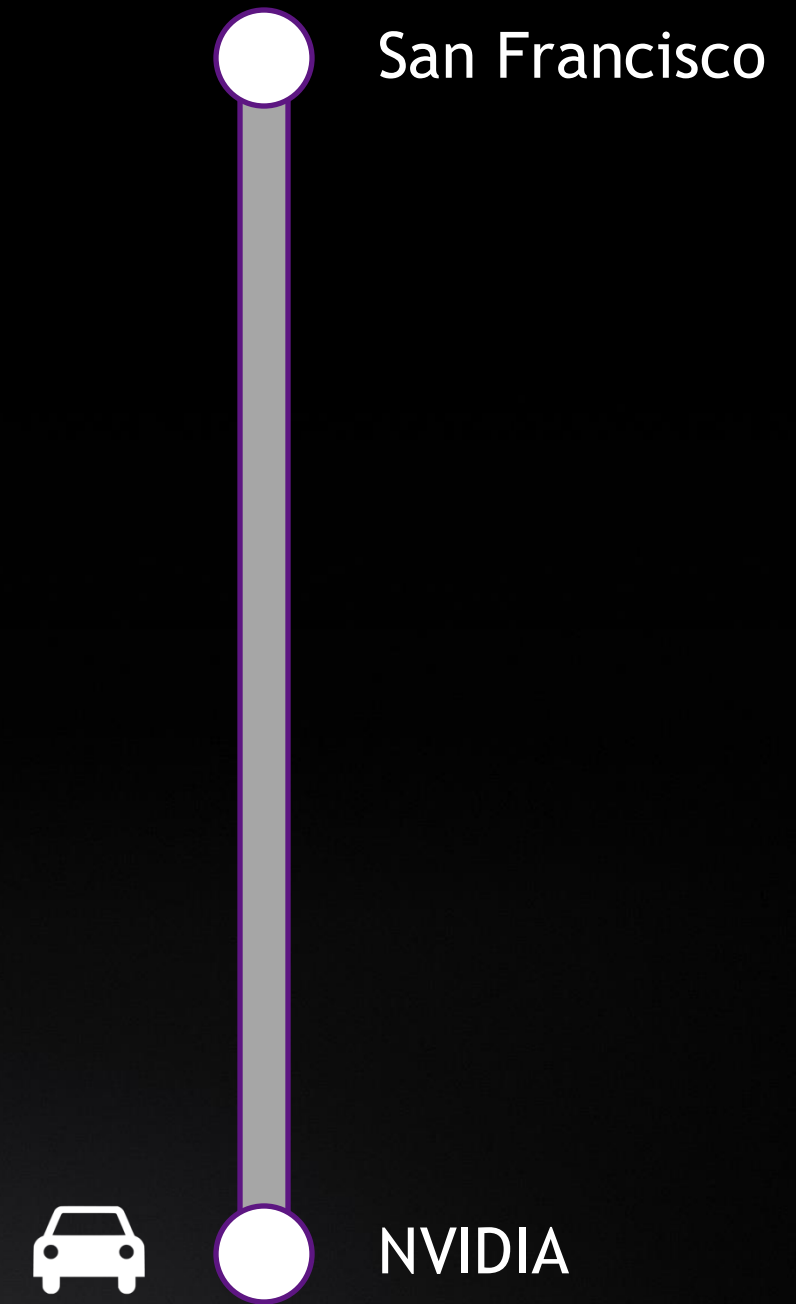
73 Minutes



45 Minutes

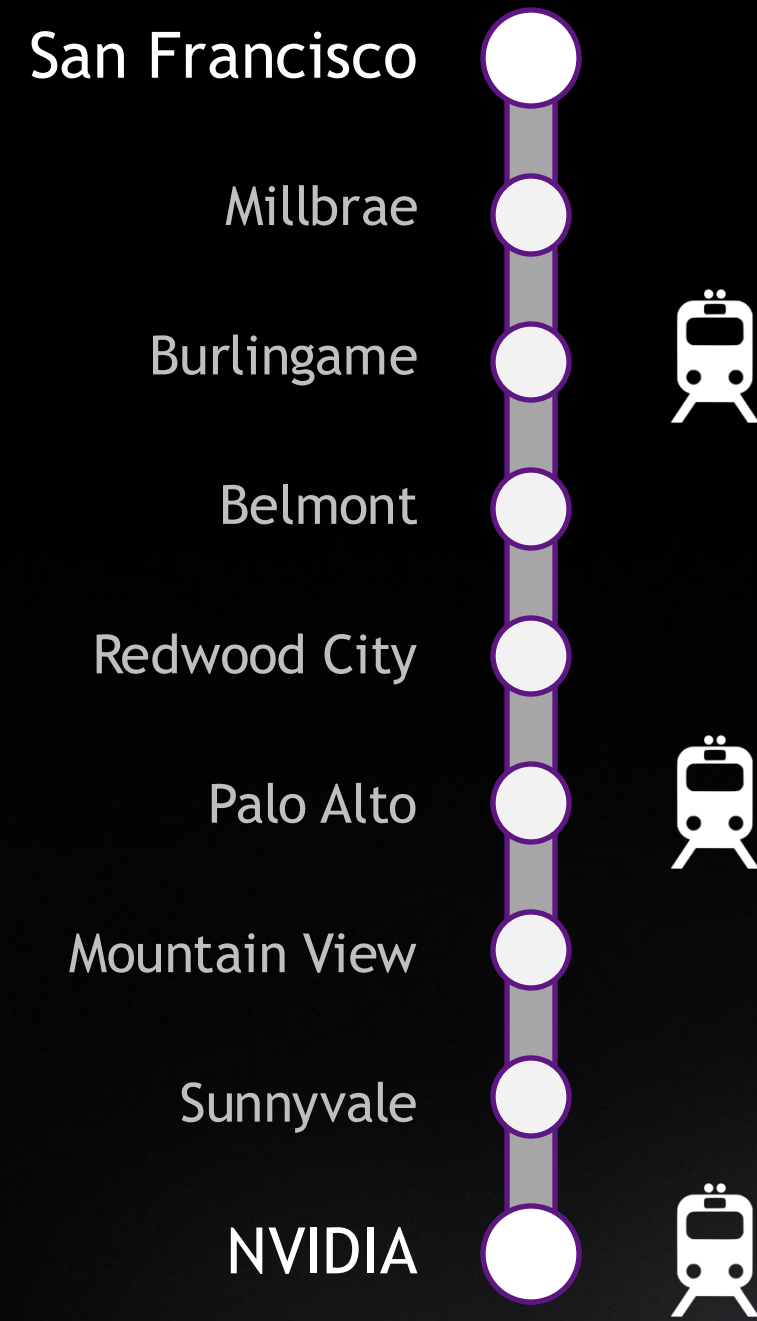


73 Minutes

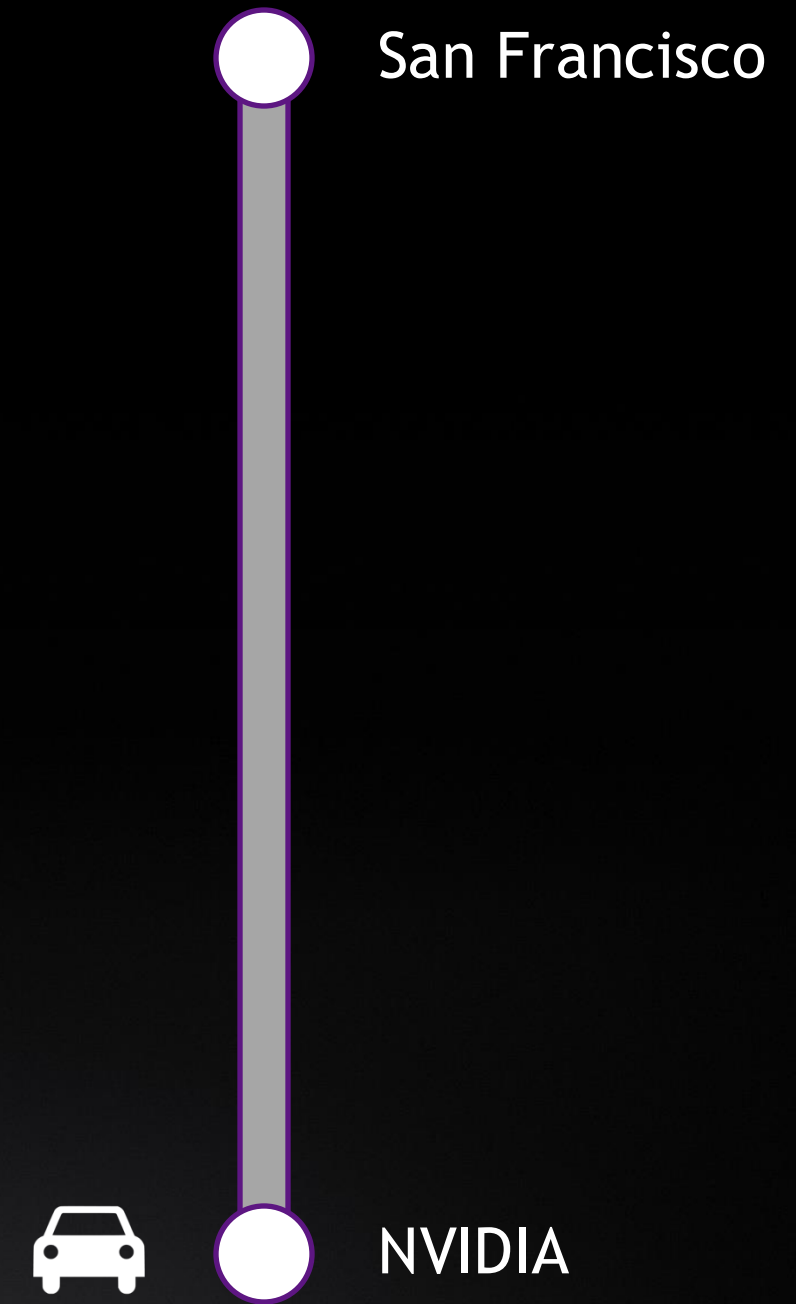


45 Minutes





73 Minutes



45 Minutes



San Francisco

Millbrae

Burlingame

Belmont

Redwood City

Palo Alto

Mountain View

Sunnyvale

NVIDIA

73 Minutes



San Francisco

NVIDIA

45 Minutes





San Francisco

Millbrae

Burlingame

Belmont

Redwood City

Palo Alto

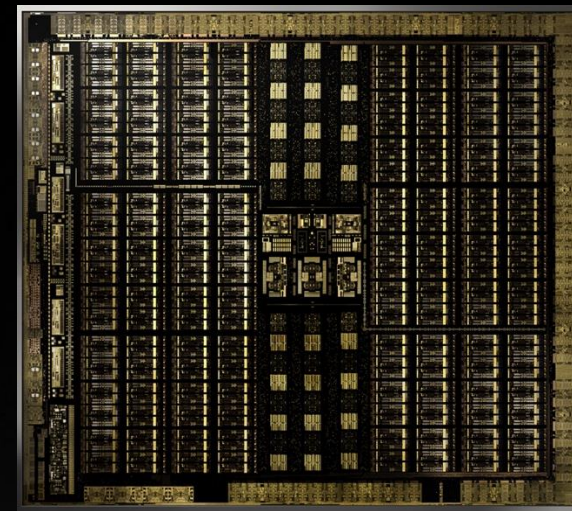
Mountain View

Sunnyvale

NVIDIA



73 Minutes

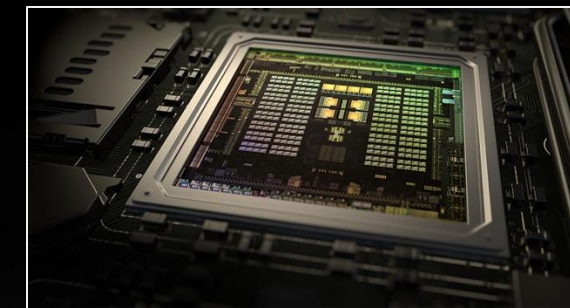


San Francisco

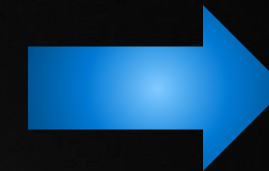
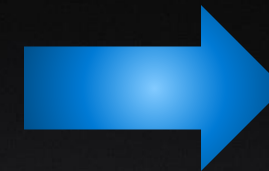
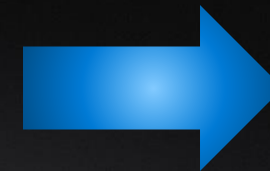
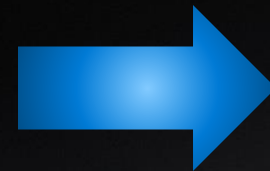
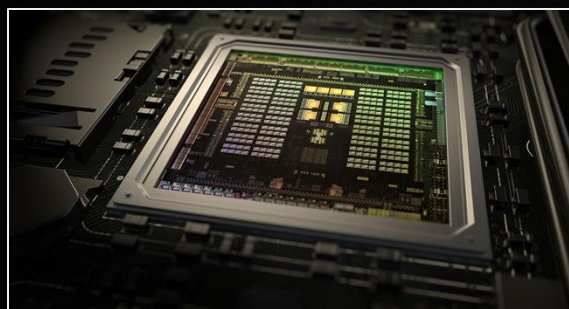
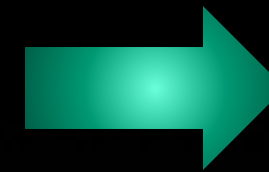
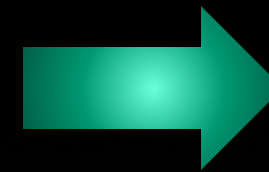
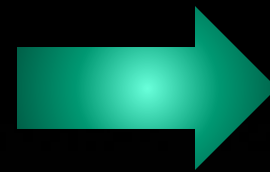
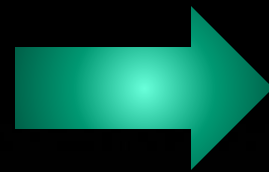
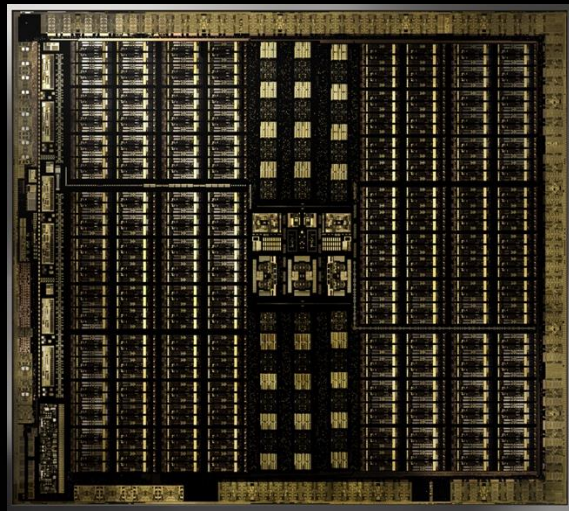


NVIDIA

45 Minutes



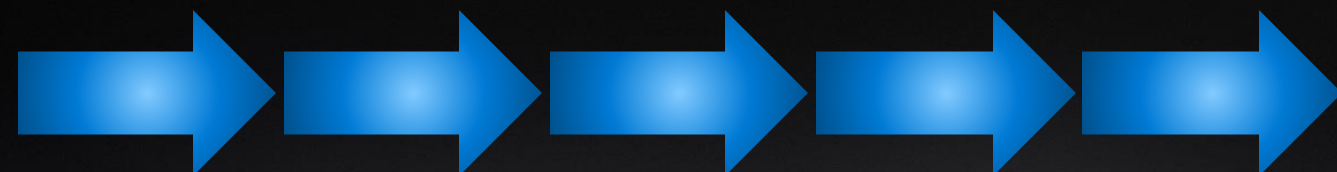
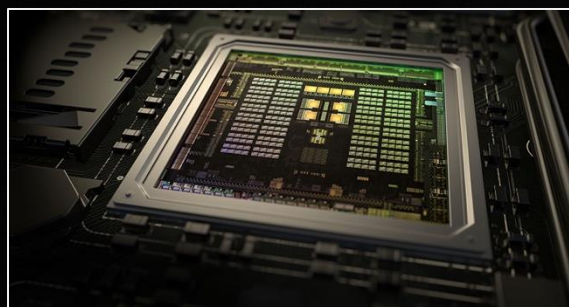
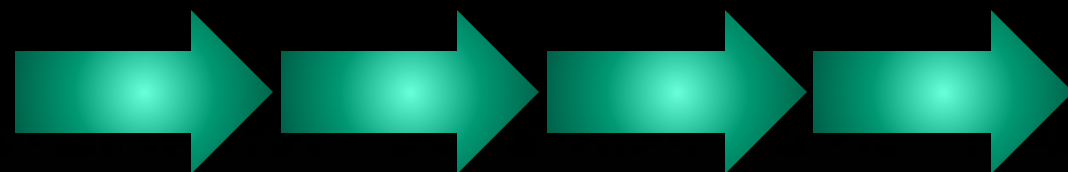
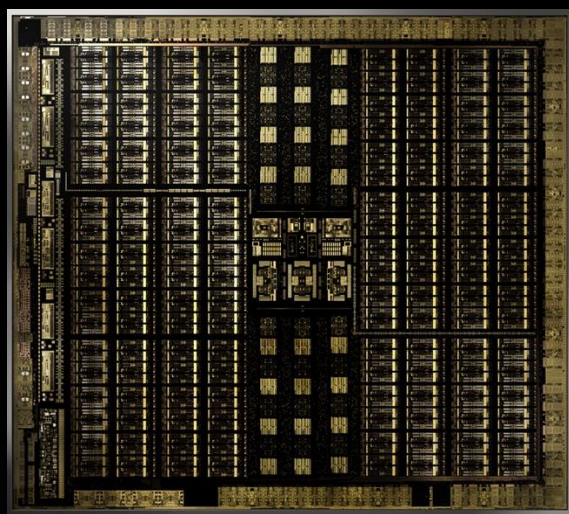
# OVERSUBSCRIPTION MEANS KEEPING THE GPU BUSY



time



# CPU/GPU ASYNCHRONY IS FUNDAMENTAL



time



Synchronous

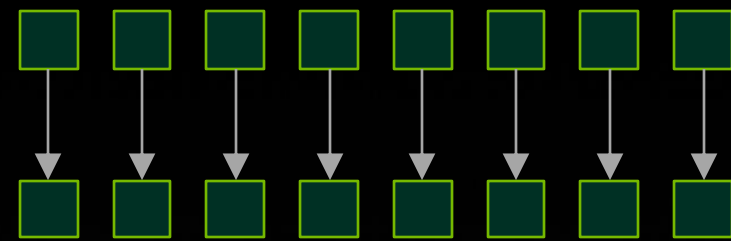


Asynchronous



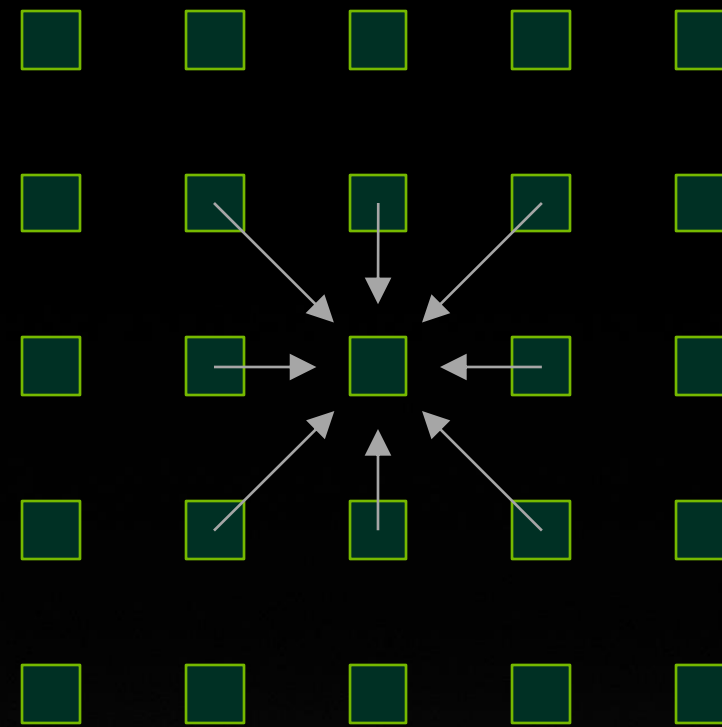
# BUT NOT ALL THREADS WANT TO WORK INDEPENDENTLY

In fact, threads are very rarely completely independent



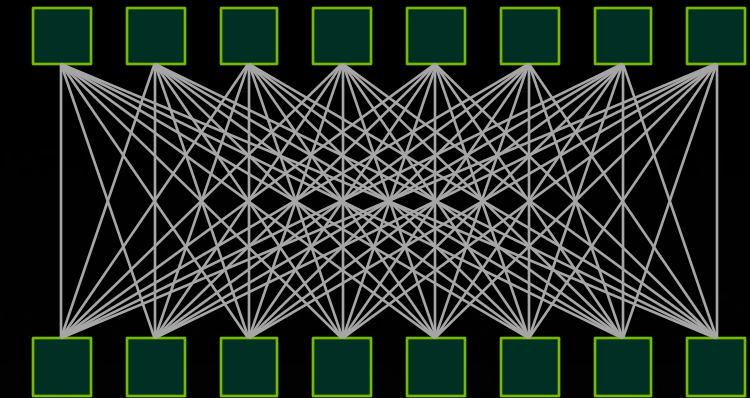
Element-wise

DAXPY



Local

Convolution



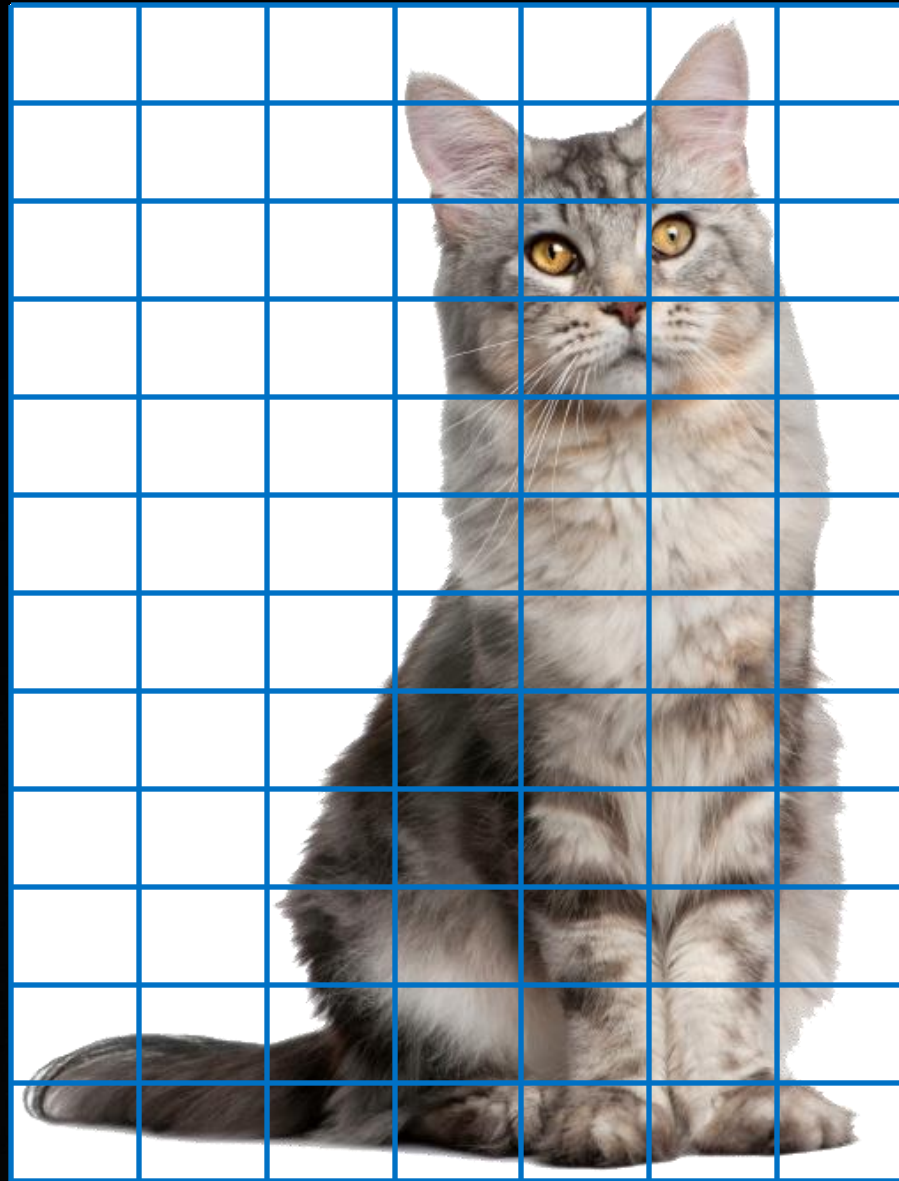
All-to-All

Fourier Transform

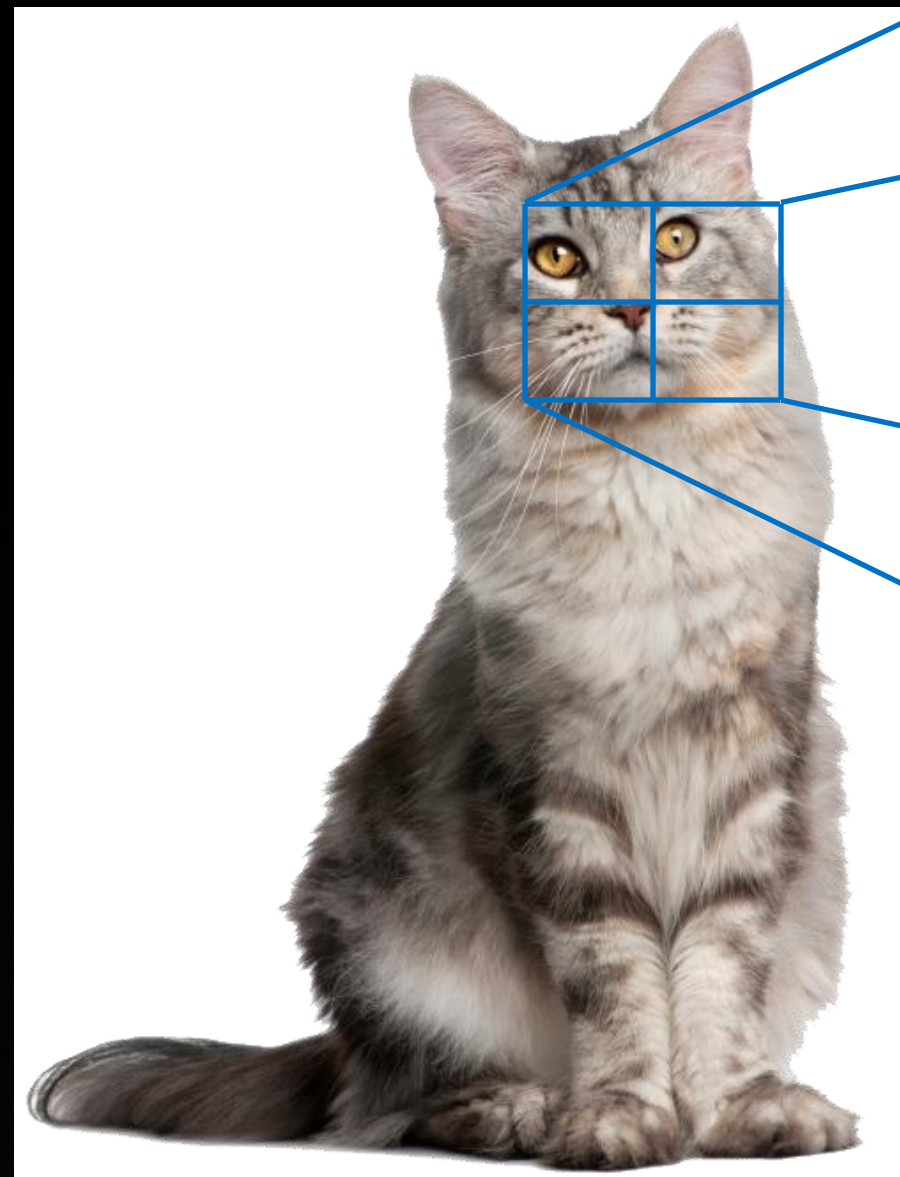








1. Overlay with a grid



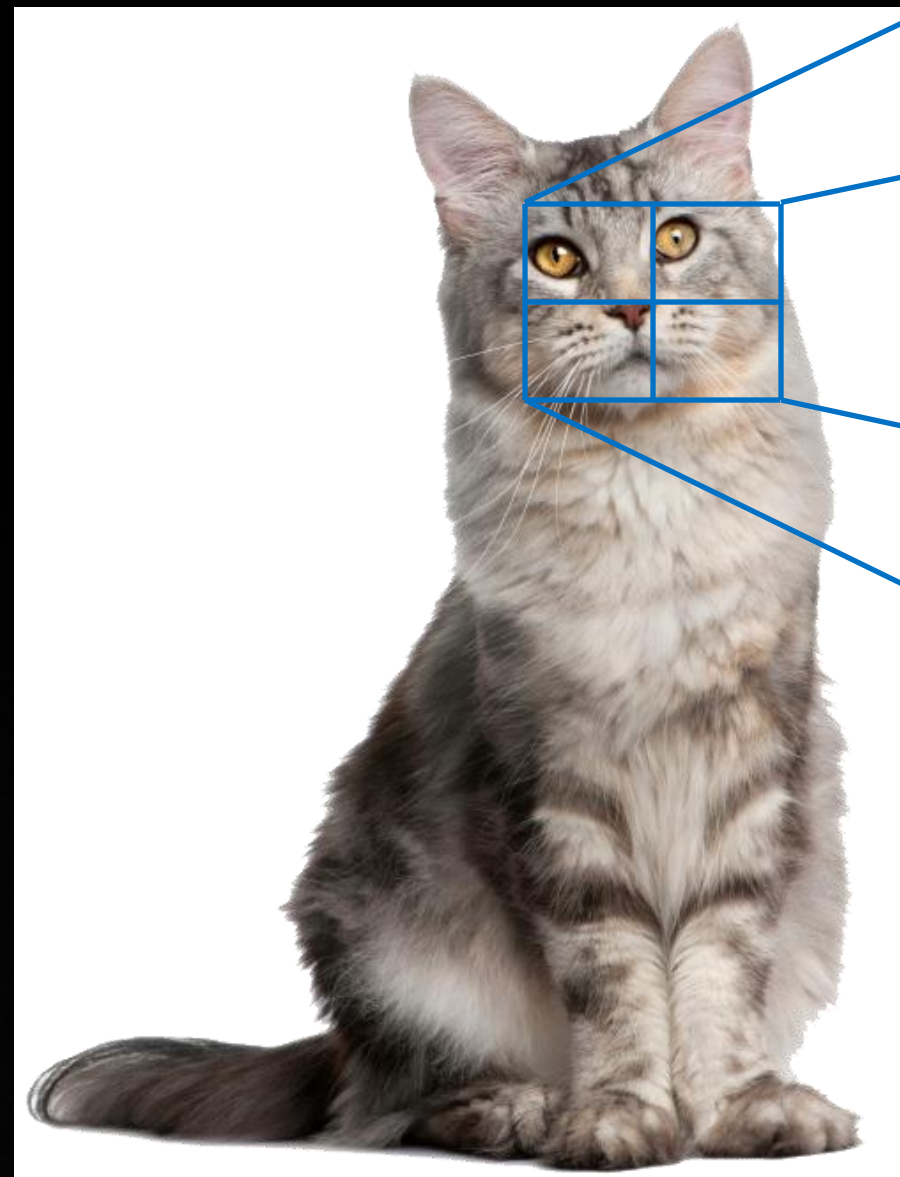
1. Overlay with a grid



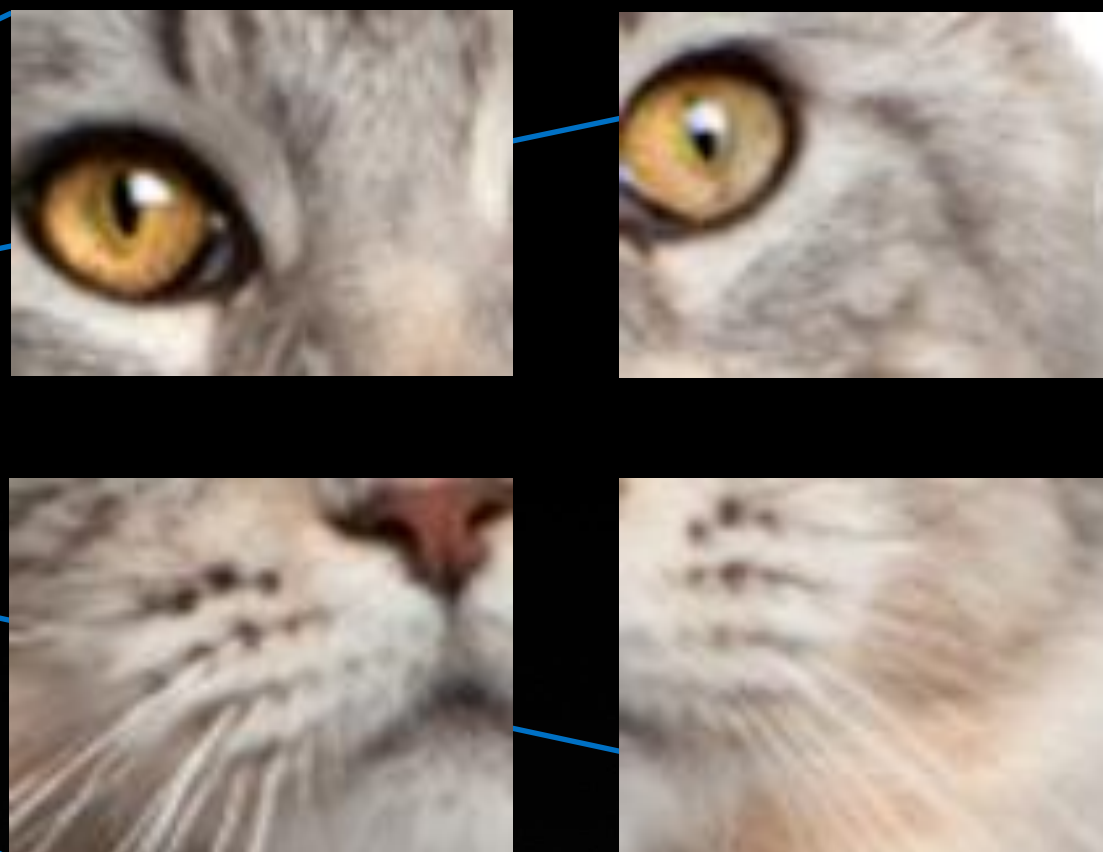
2. Operate on blocks within the grid

Blocks execute **independently**  
GPU is **oversubscribed** with blocks



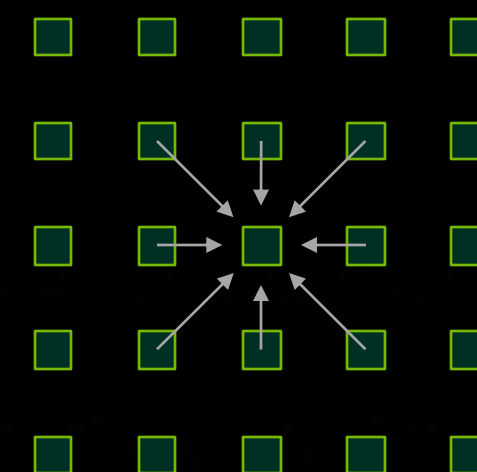


1. Overlay with a grid

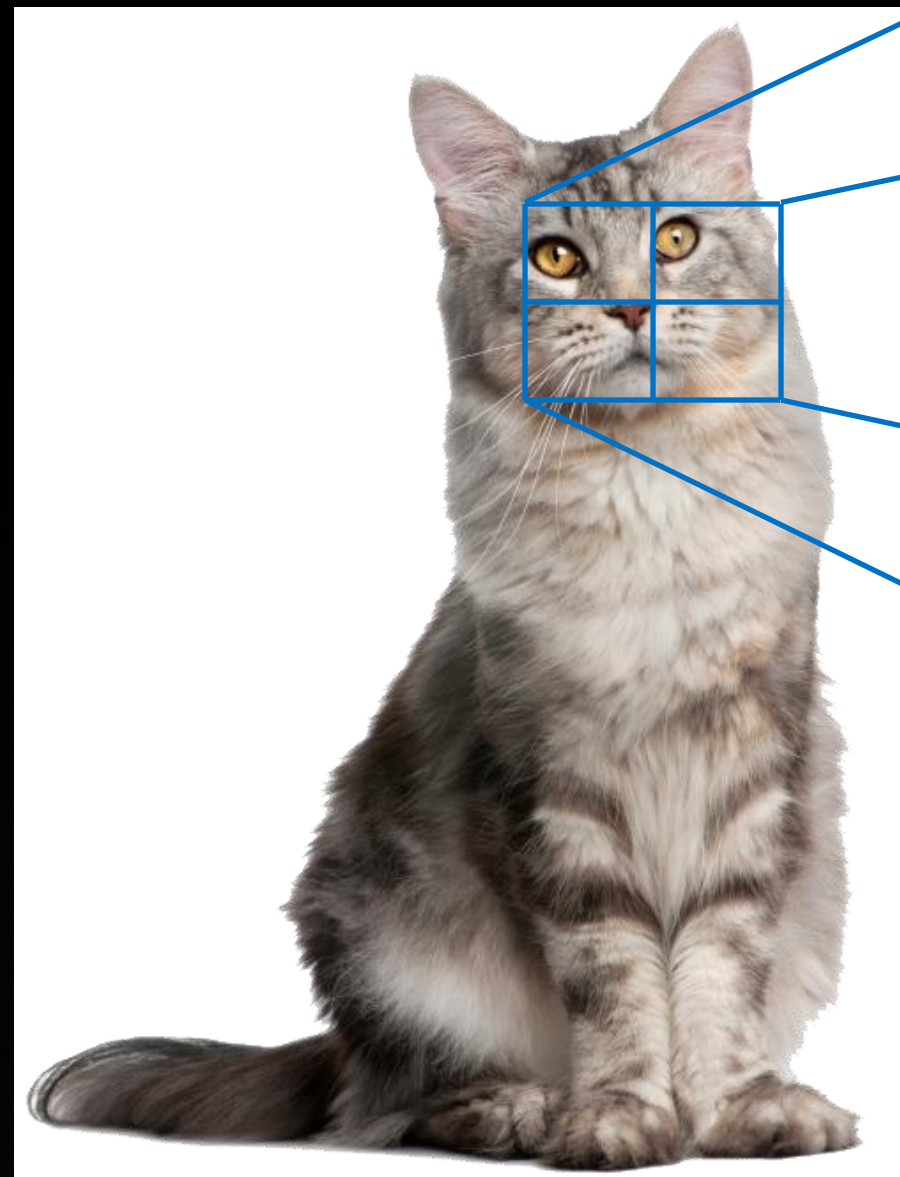


2. Operate on blocks within the grid

Blocks execute **independently**  
GPU is **oversubscribed** with blocks



3. Many threads work together in each block for **local** data sharing

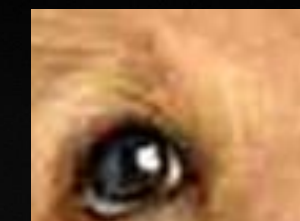
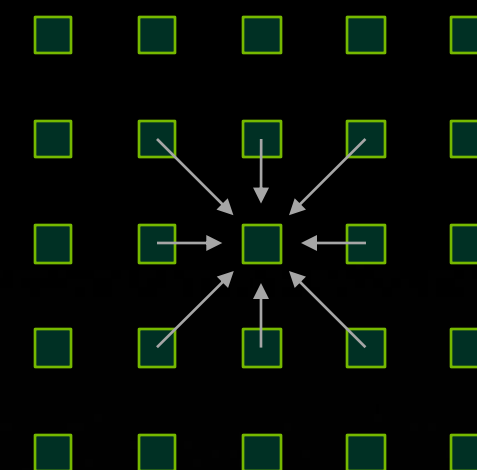


1. Overlay with a grid



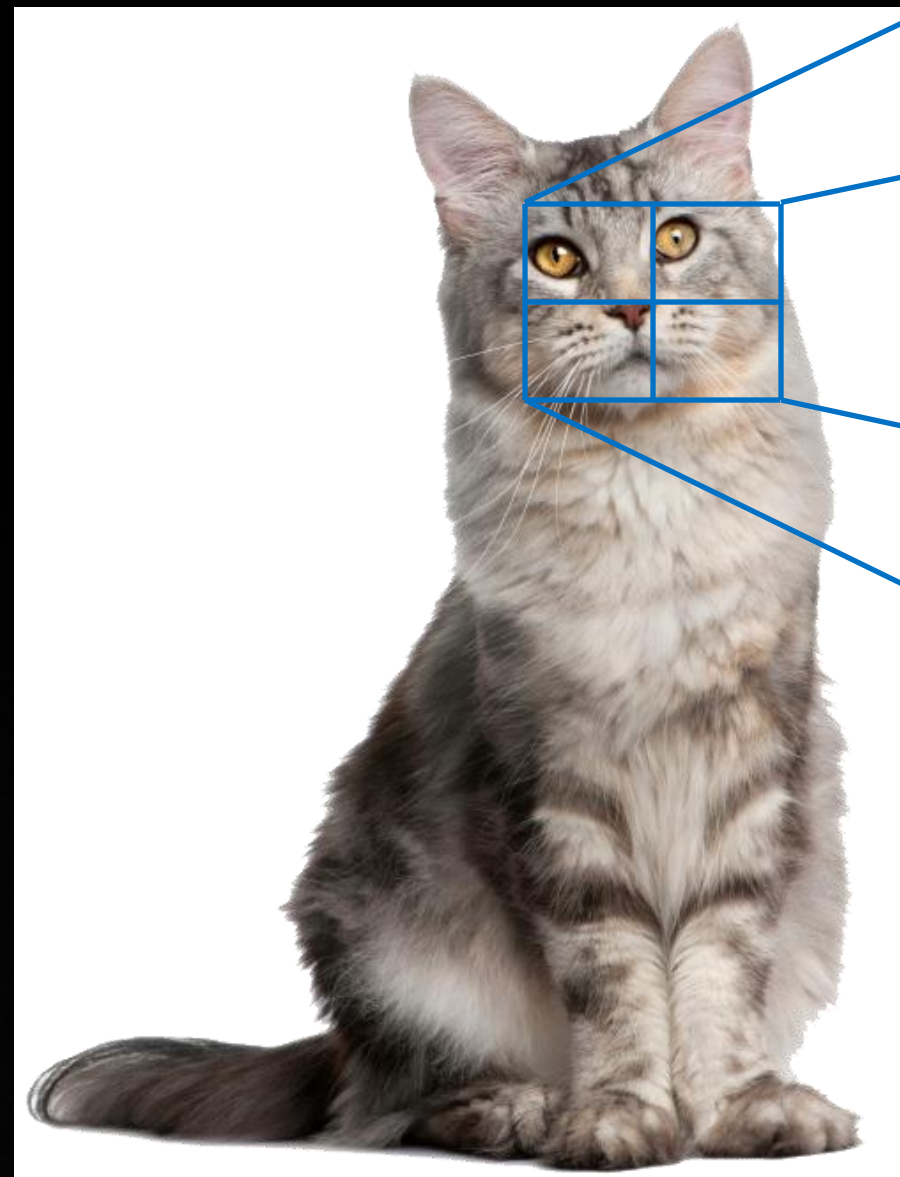
2. Operate on blocks within the grid

Blocks execute **independently**  
GPU is **oversubscribed** with blocks



3. Many threads work together in each block for **local** data sharing



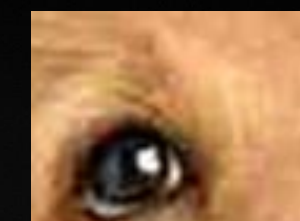
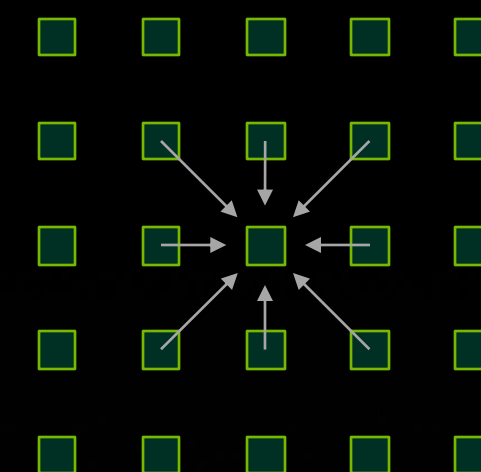


1. Overlay with a grid

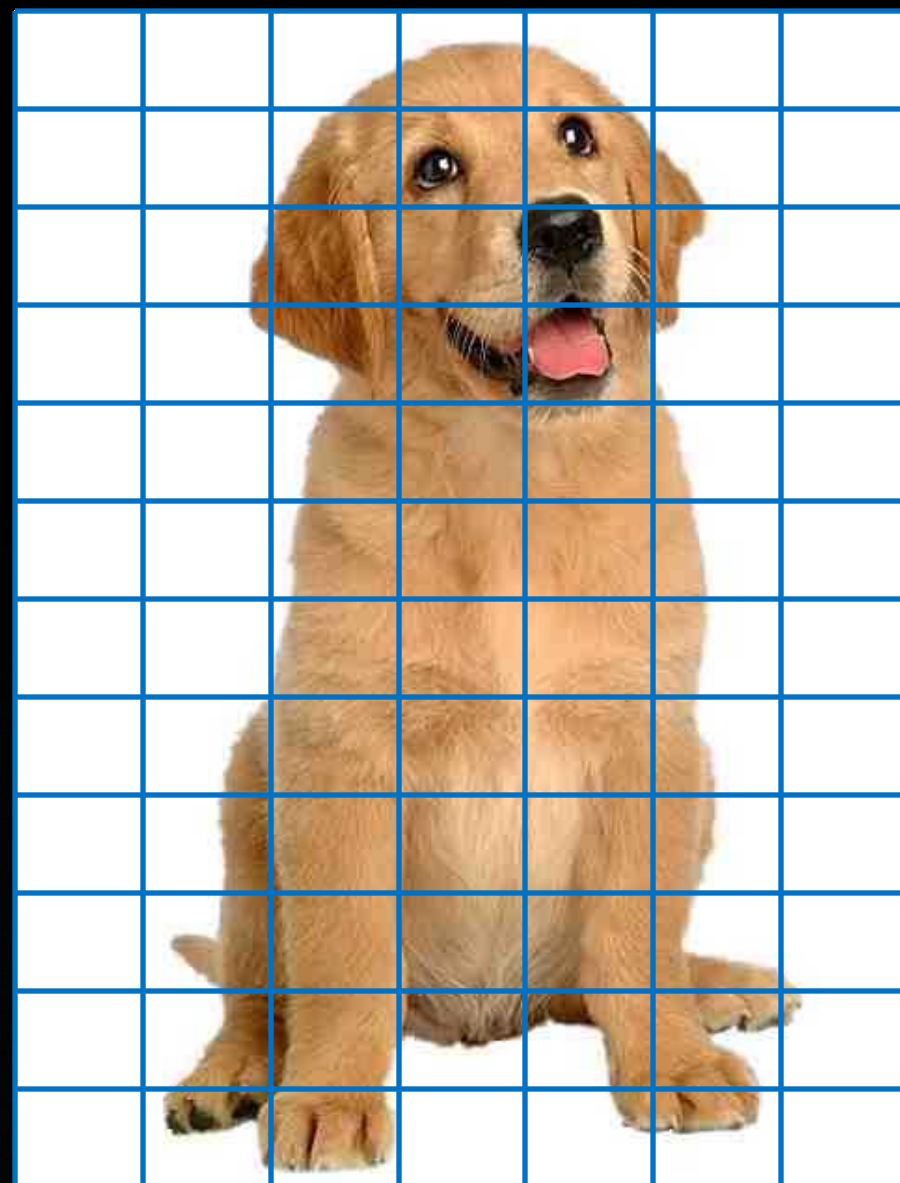


2. Operate on blocks within the grid

Blocks execute **independently**  
GPU is **oversubscribed** with blocks



3. Many threads work together in each block for **local** data sharing

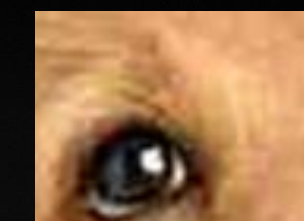
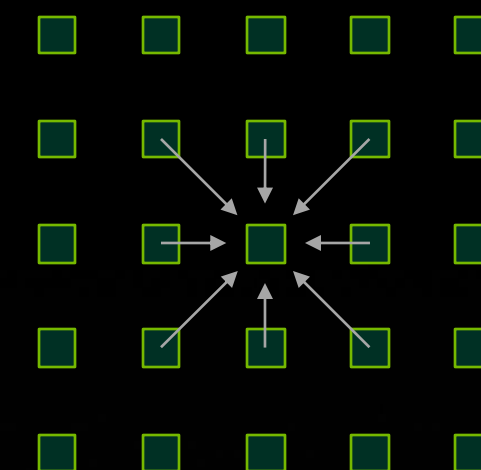


1. Overlay with a grid



2. Operate on blocks within the grid

Blocks execute **independently**  
GPU is **oversubscribed** with blocks



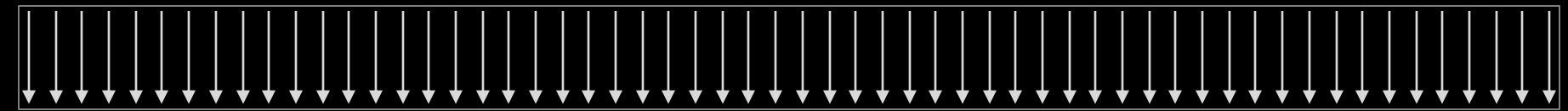
3. Many threads work together in each block for **local** data sharing



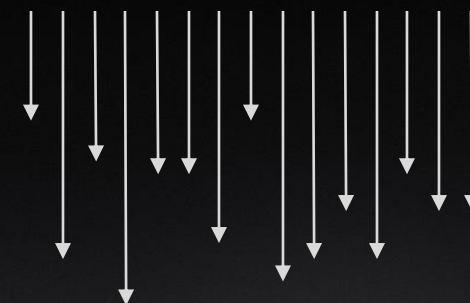
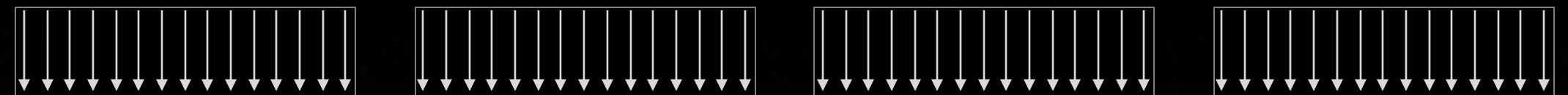
# CUDA'S HIERARCHICAL EXECUTION MODEL



A grid represents all work to be done

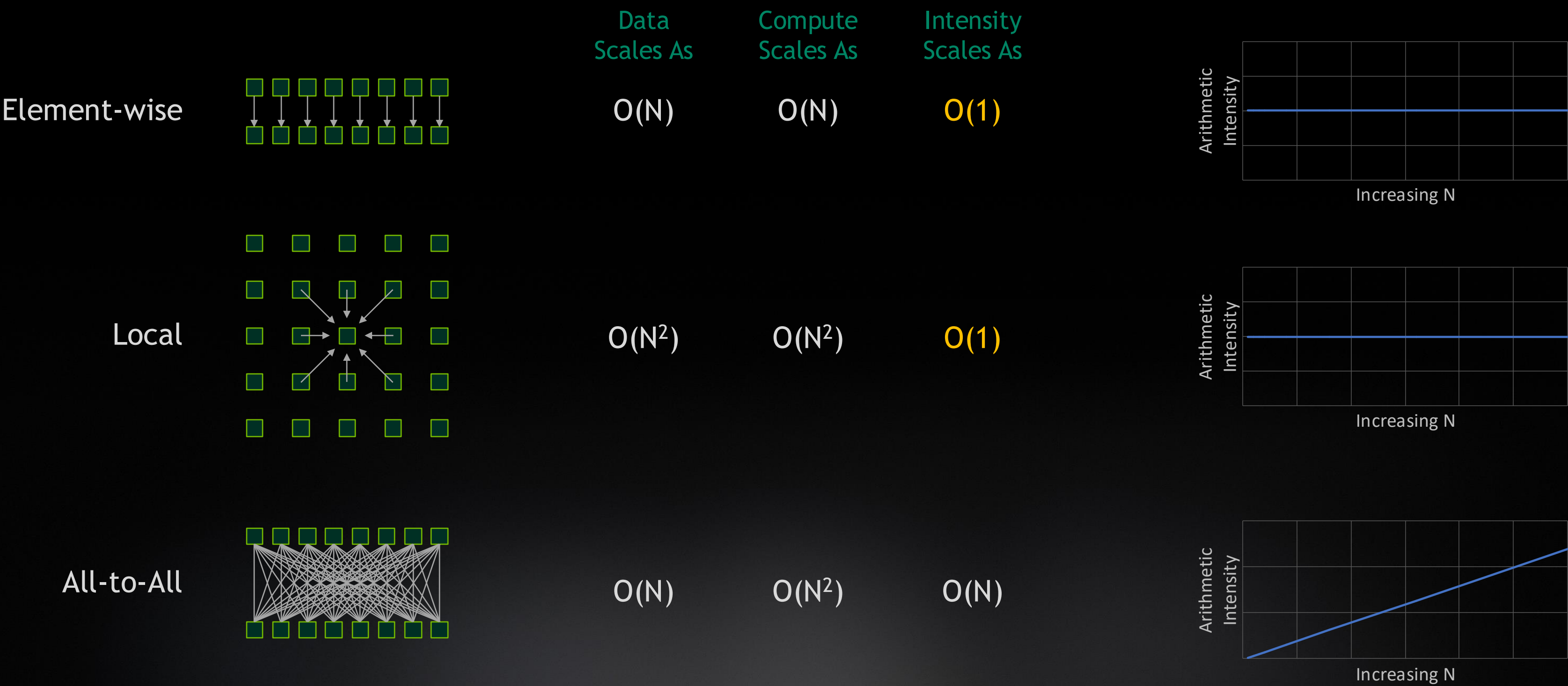


The grid comprises many blocks with an equal number of threads



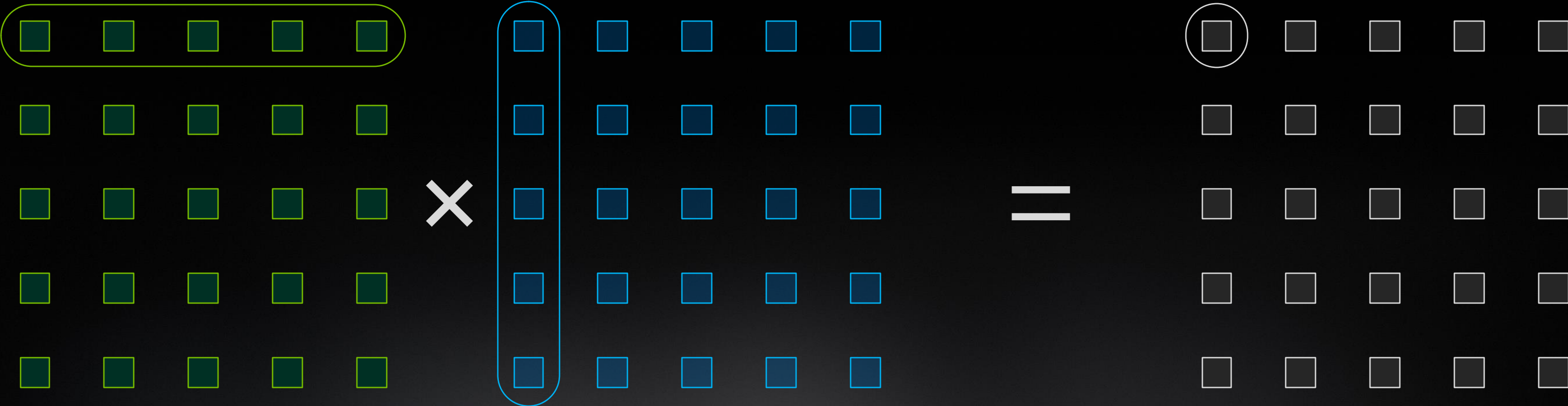
Threads within a block run **independently** but may **synchronize** to exchange data

# BEATING COMPUTE INTENSITY IS ALL ABOUT SCALING

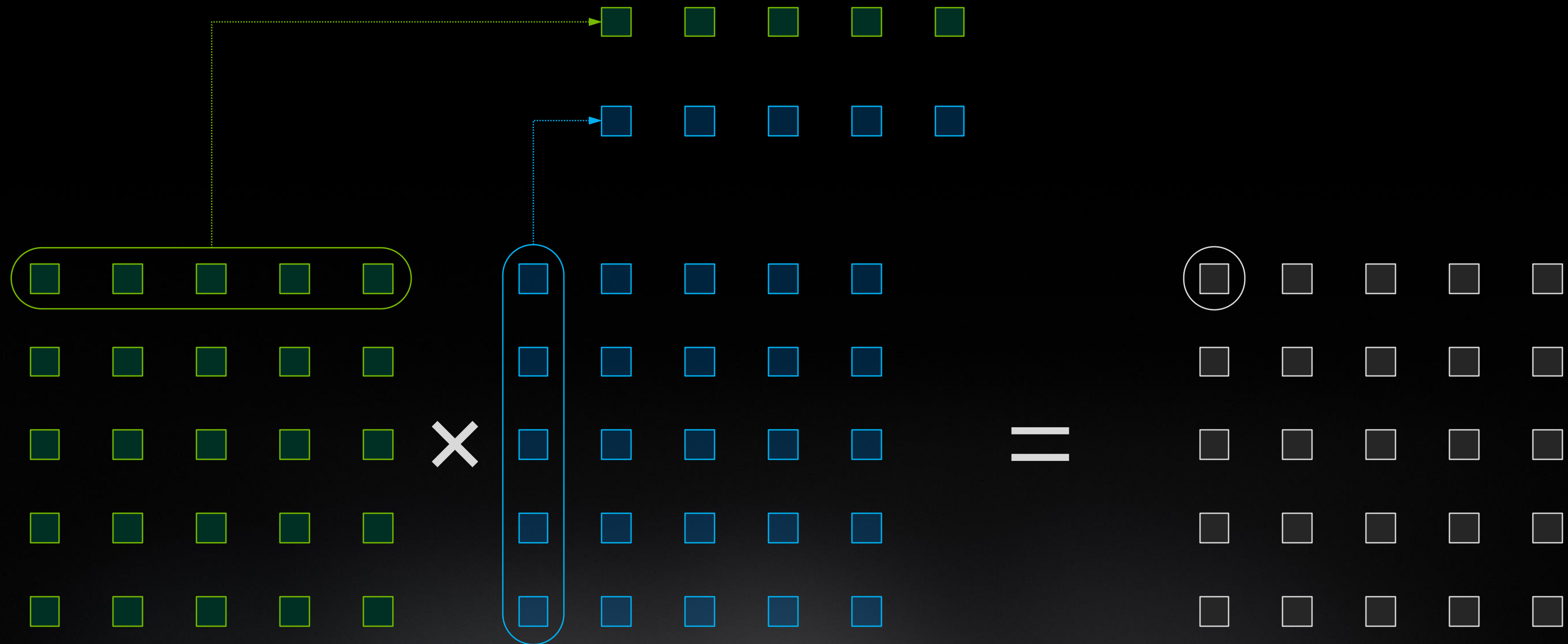




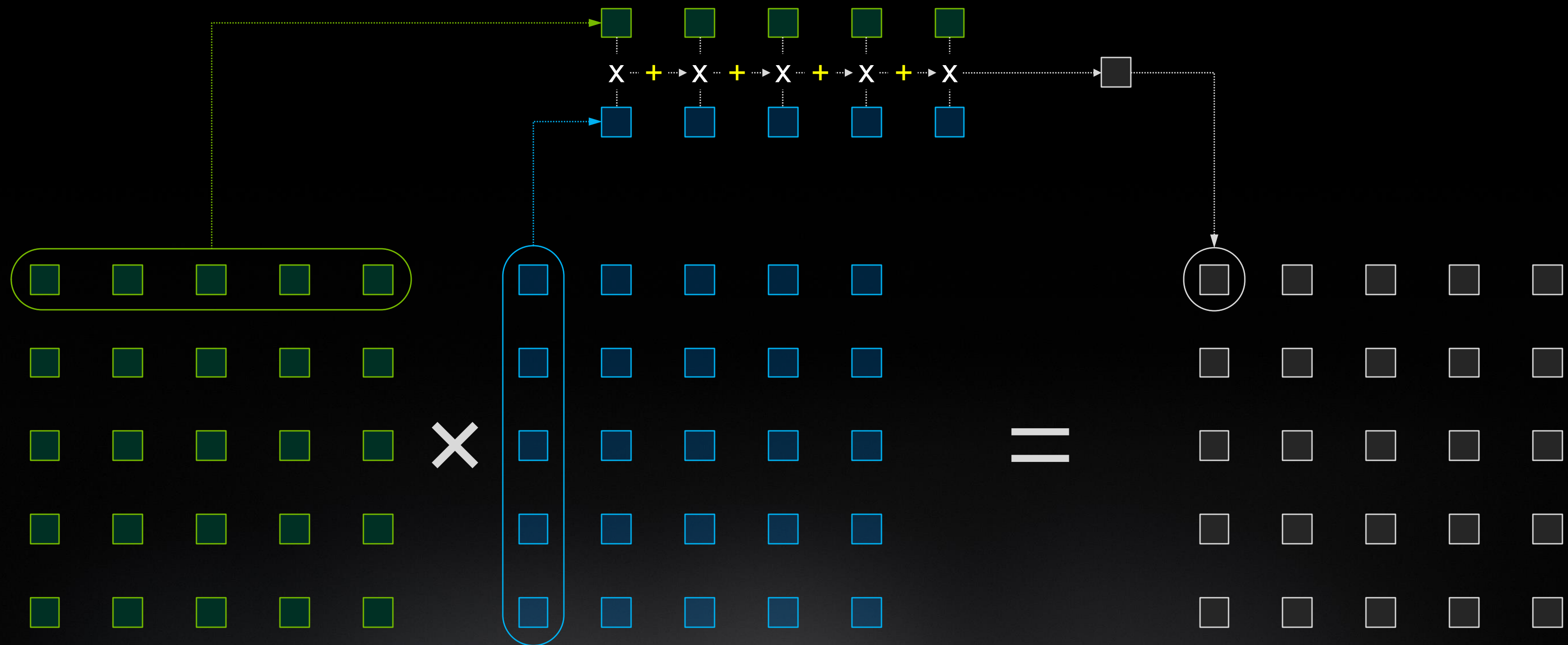
# MATRIX MULTIPLICATION



# MATRIX MULTIPLICATION

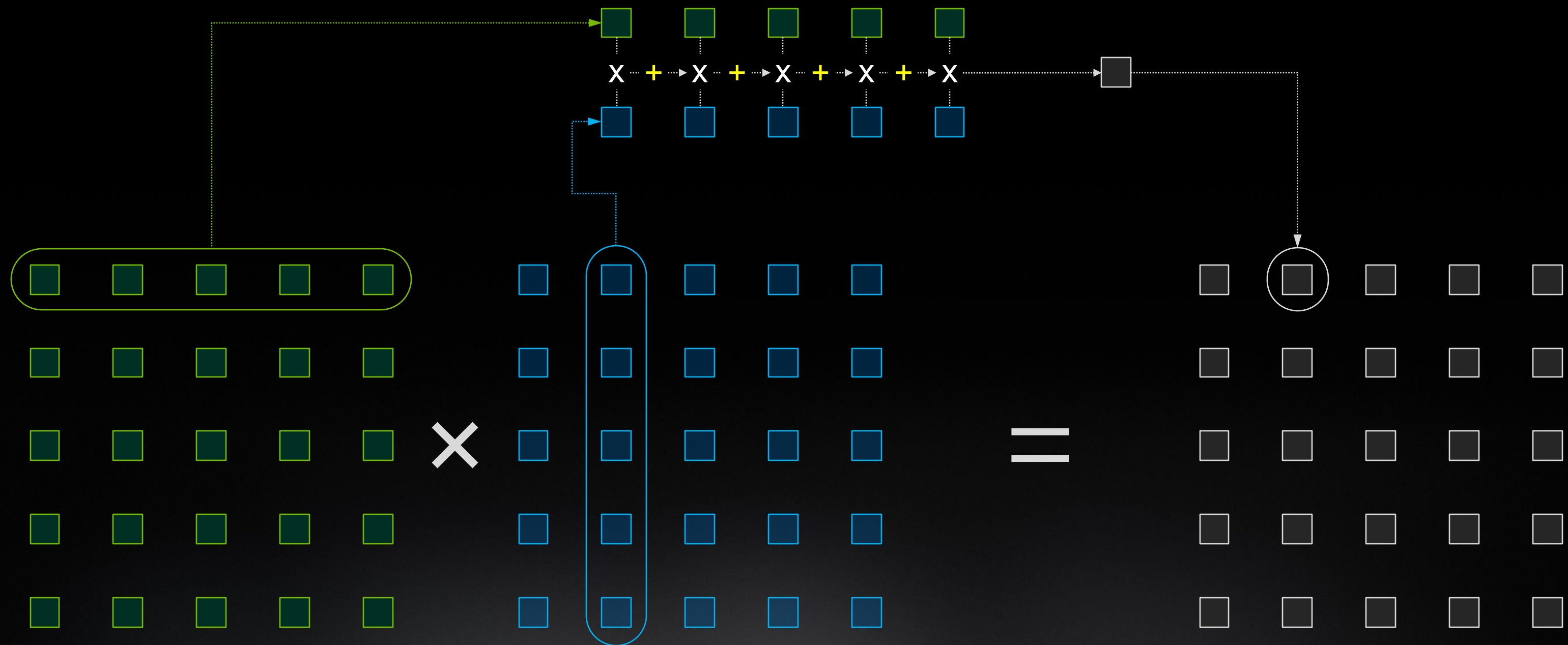


# MATRIX MULTIPLICATION

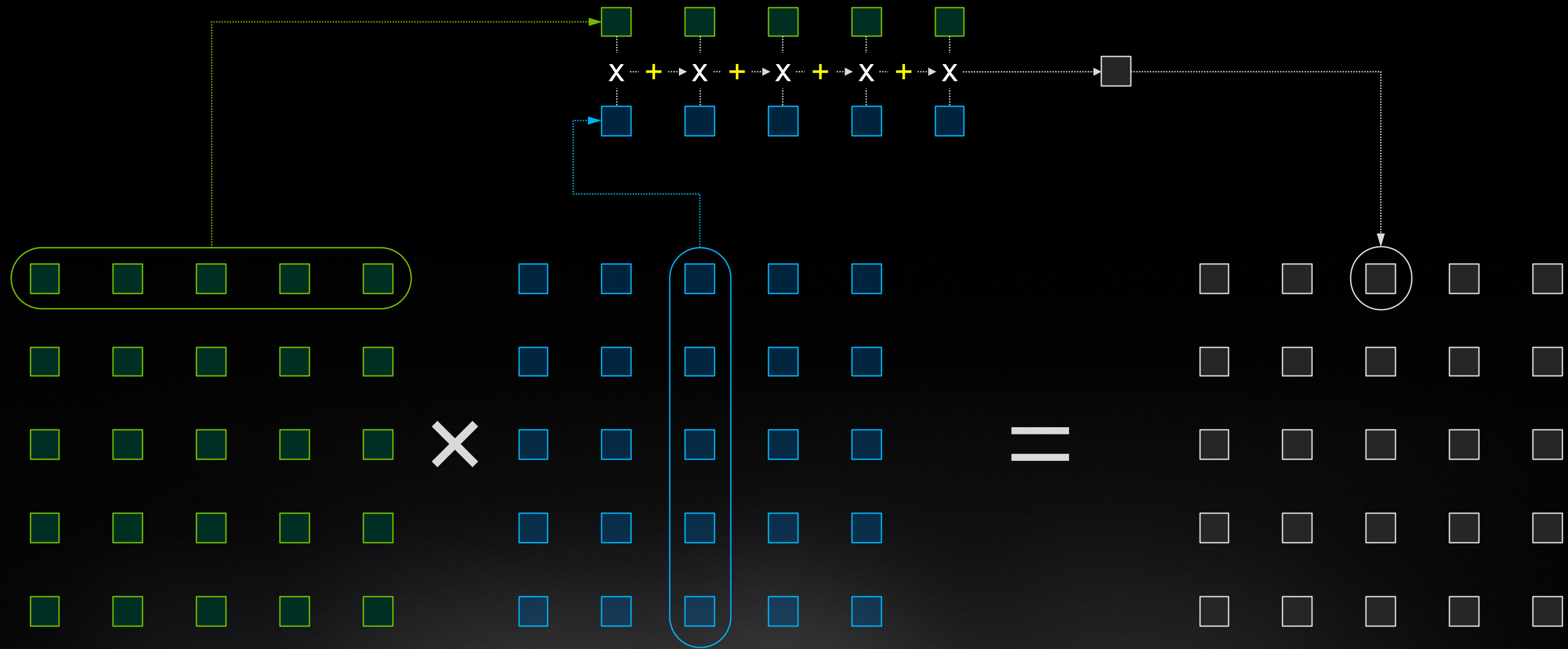




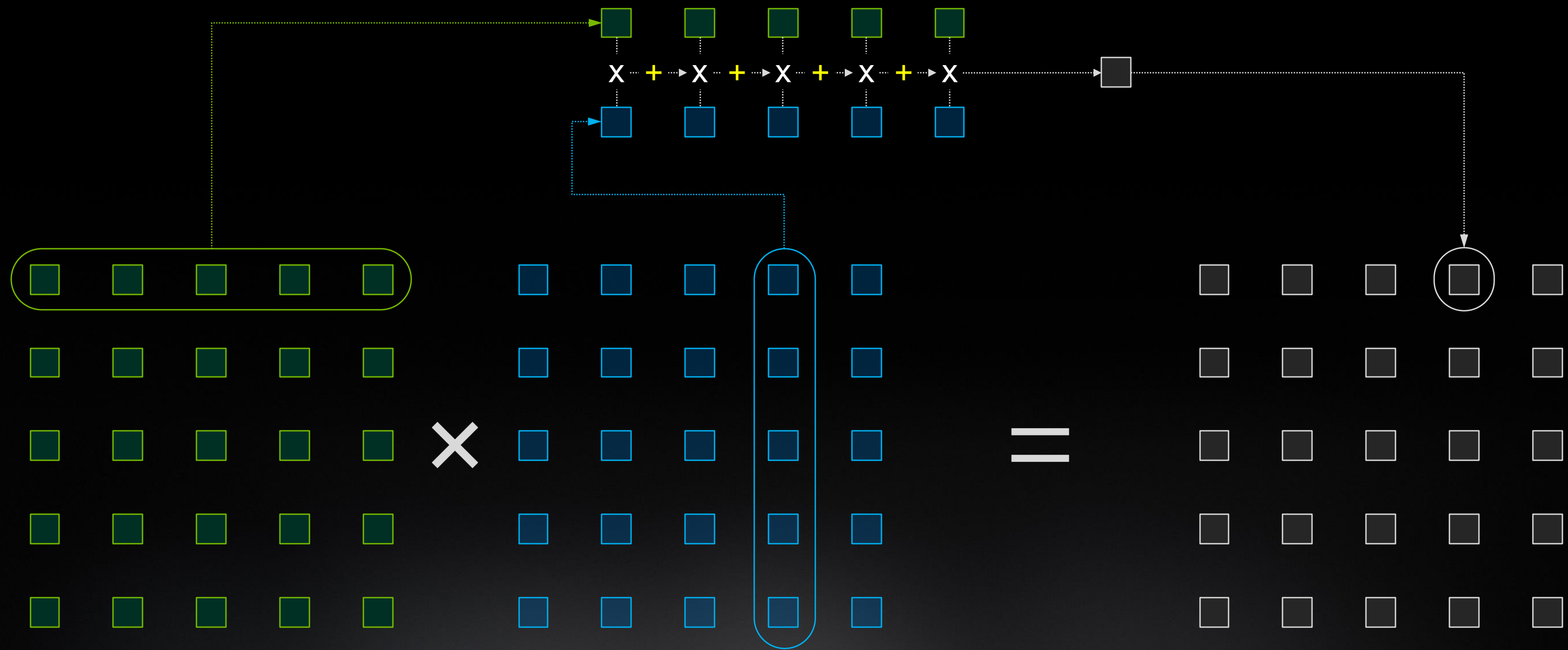
# MATRIX MULTIPLICATION



# MATRIX MULTIPLICATION

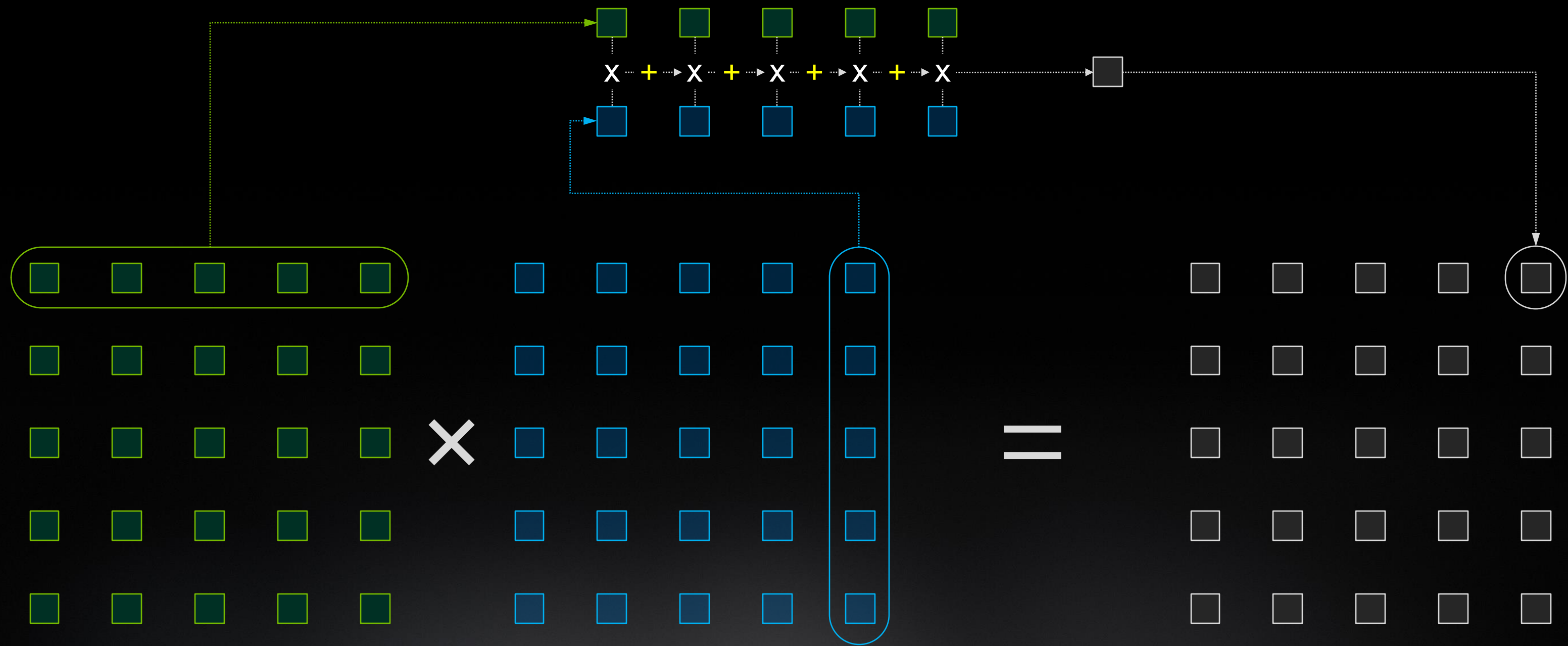


# MATRIX MULTIPLICATION

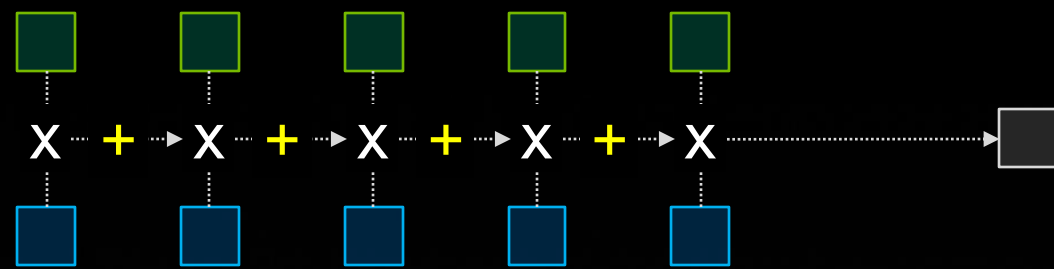




# MATRIX MULTIPLICATION



# ARITHMETIC INTENSITY OF MATRIX MULTIPLICATION



For an  $N \times N$  matrix

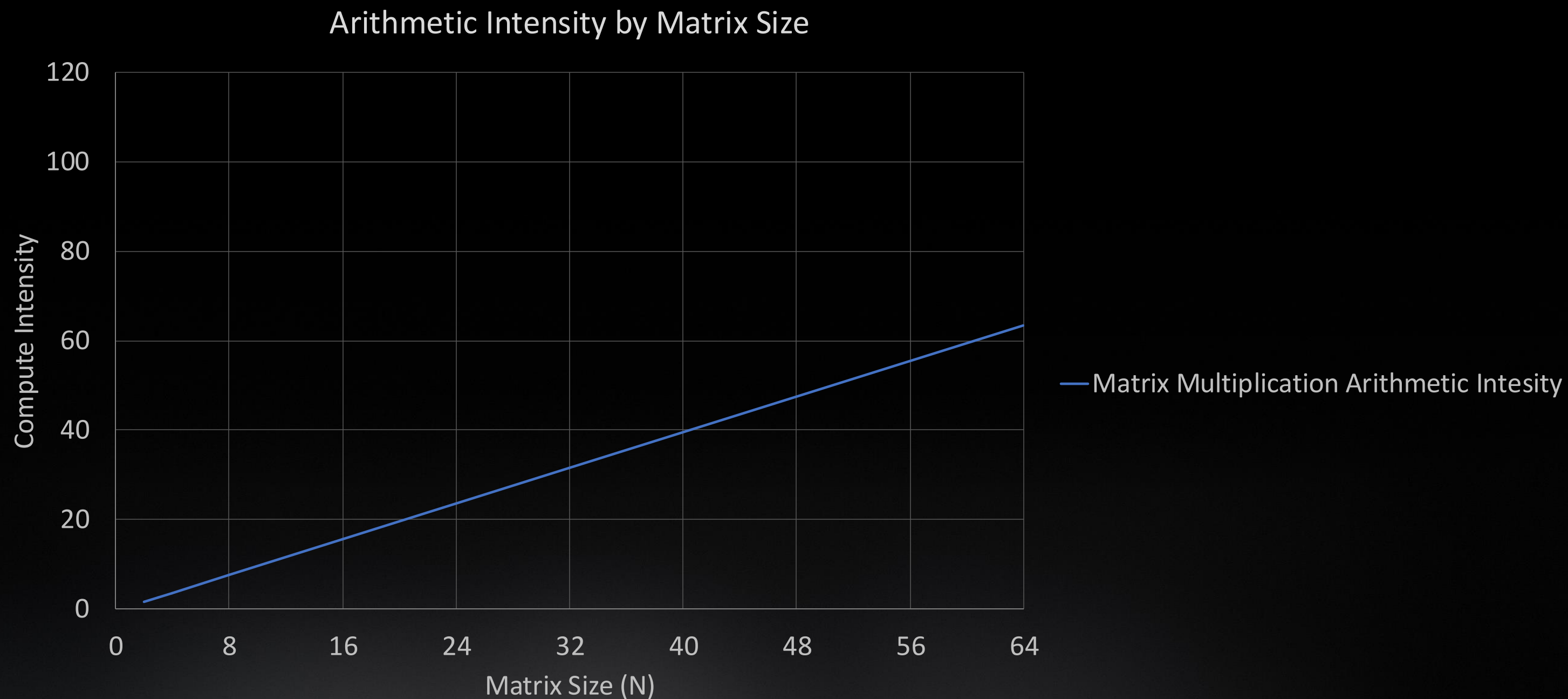
- $N$  row elements multiply with  $N$  column elements
- $N$  additions create the final result
- This is done  $N^2$  times, once for each result element

Arithmetic complexity is therefore:  $N^2 \times (2N) = 2N^3 = O(N^3)$

Number of data loads:  $O(N^2)$

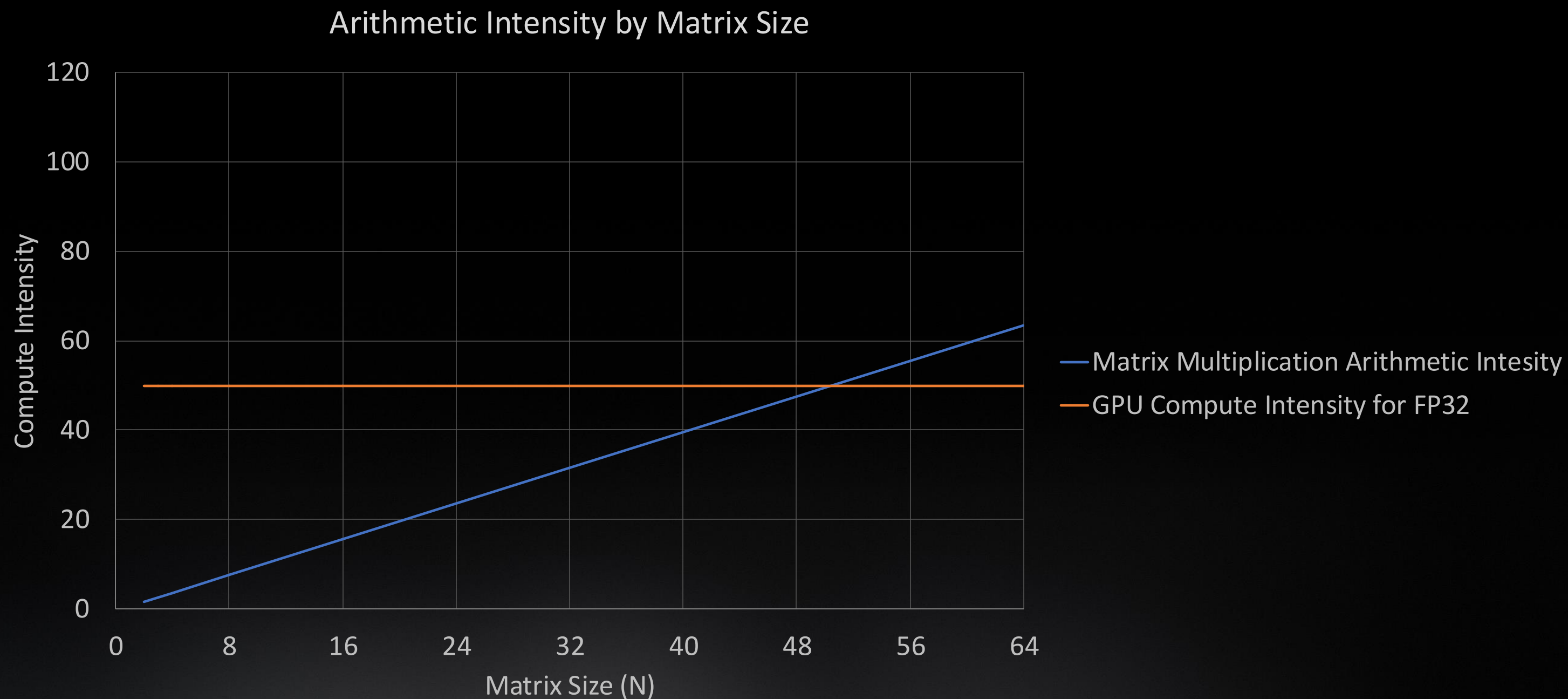
Arithmetic intensity scales as:  $N^3 / N^2 = O(N)$

# ALGORITHMIC EFFICIENCY OF MMA

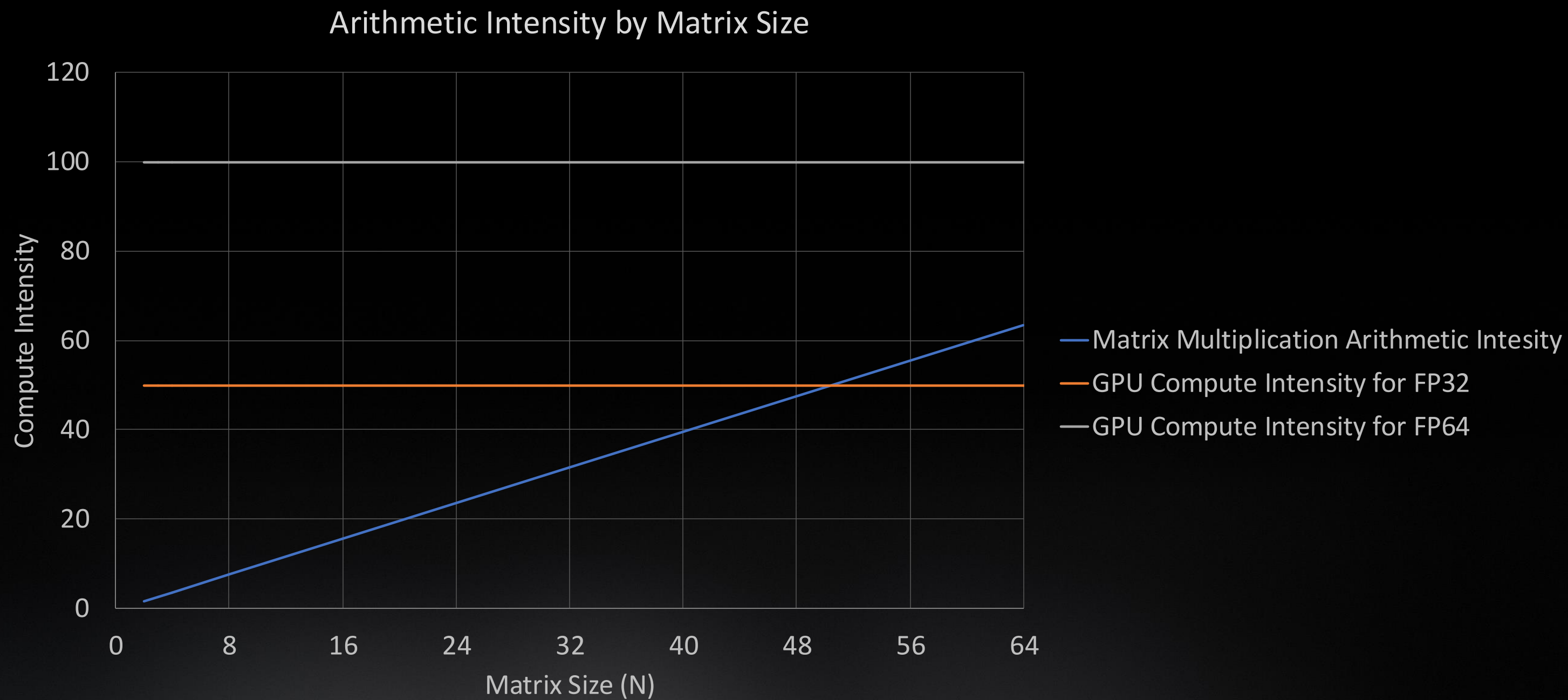




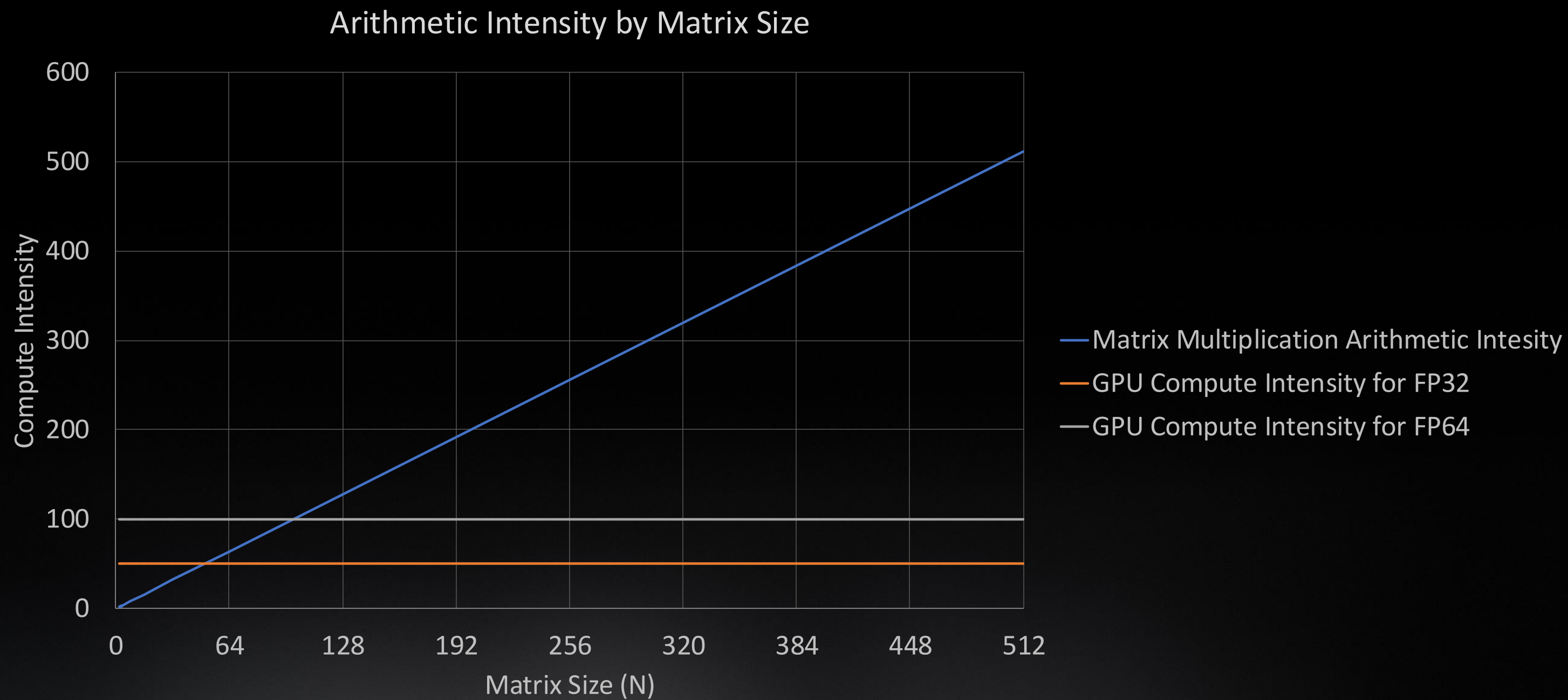
# ALGORITHMIC EFFICIENCY OF MMA



# ALGORITHMIC EFFICIENCY OF MMA

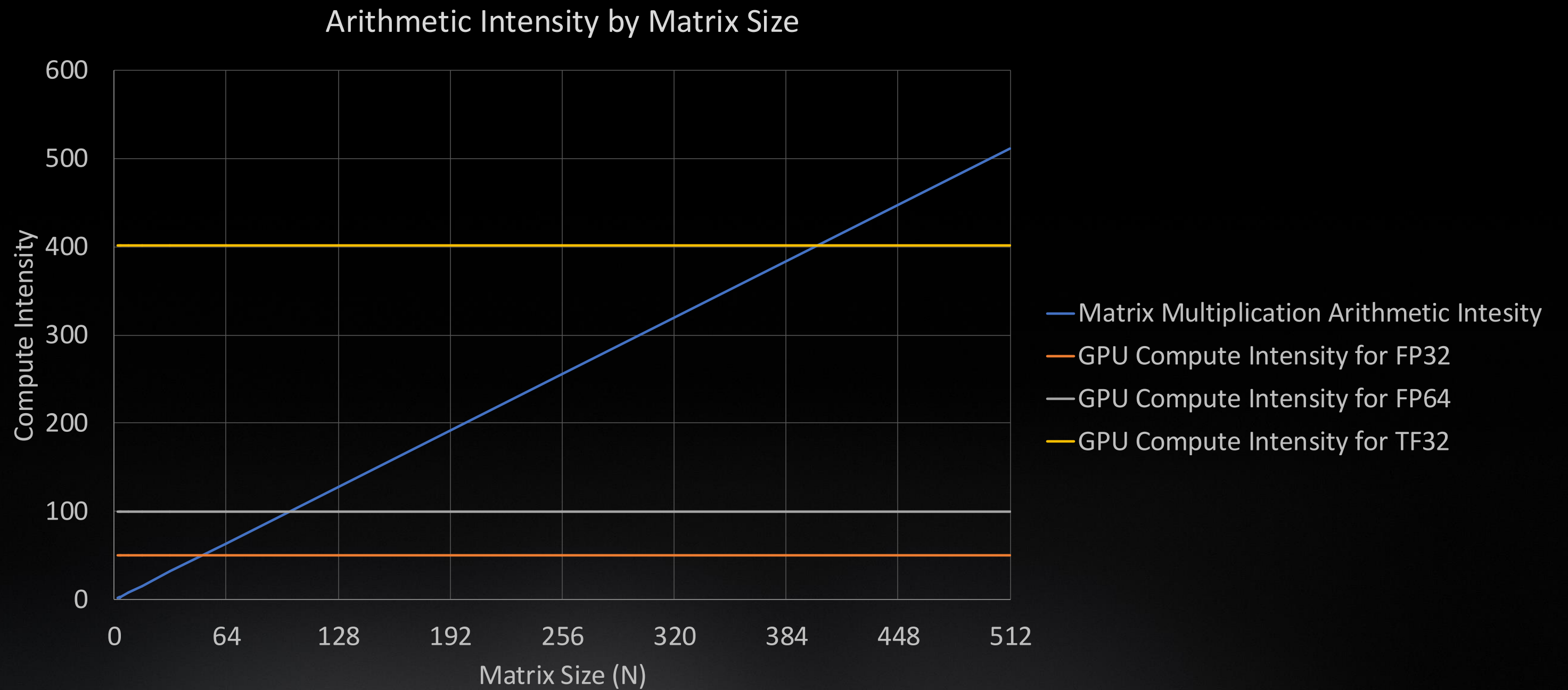


# ALGORITHMIC EFFICIENCY OF MMA



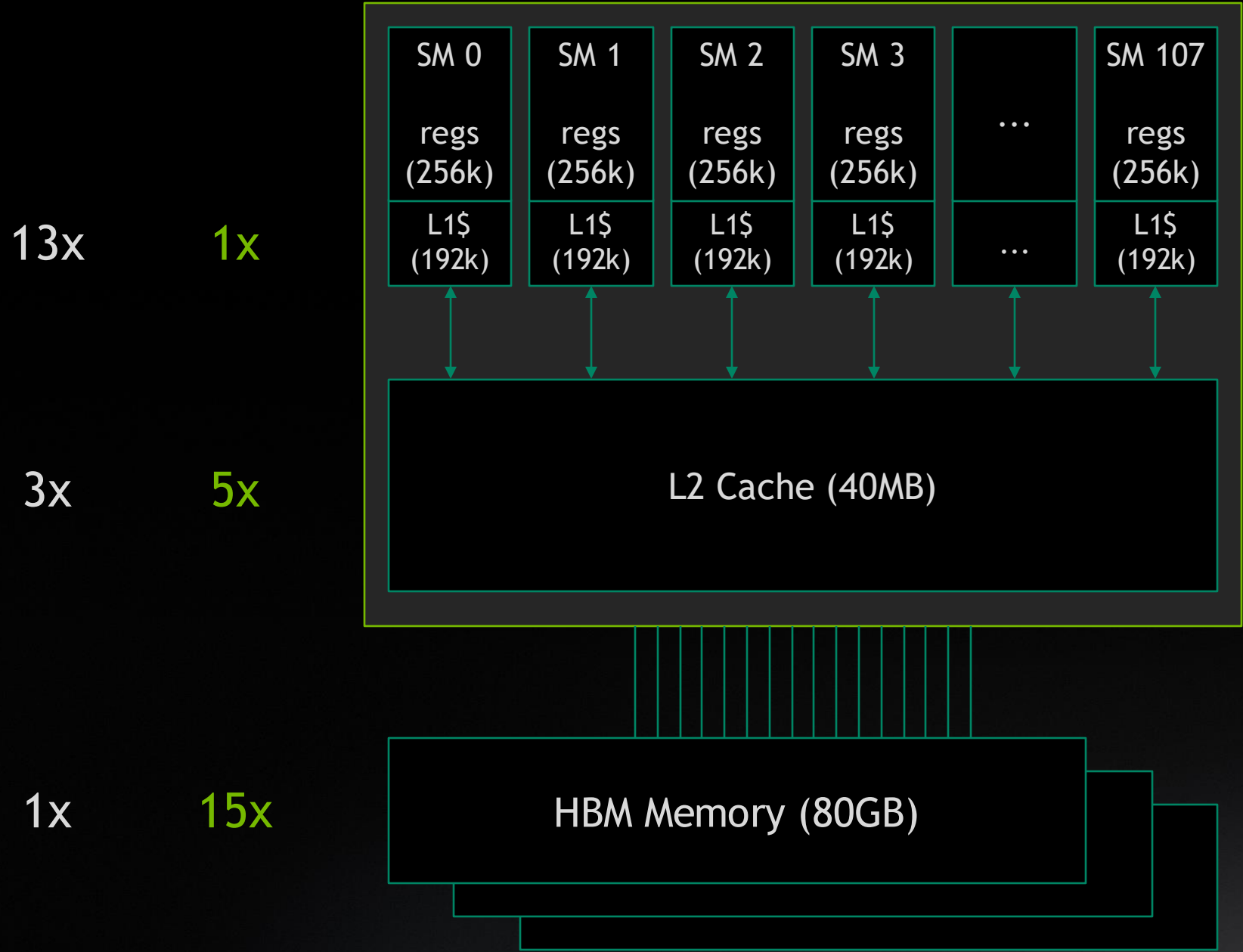


# ALGORITHMIC EFFICIENCY OF MMA



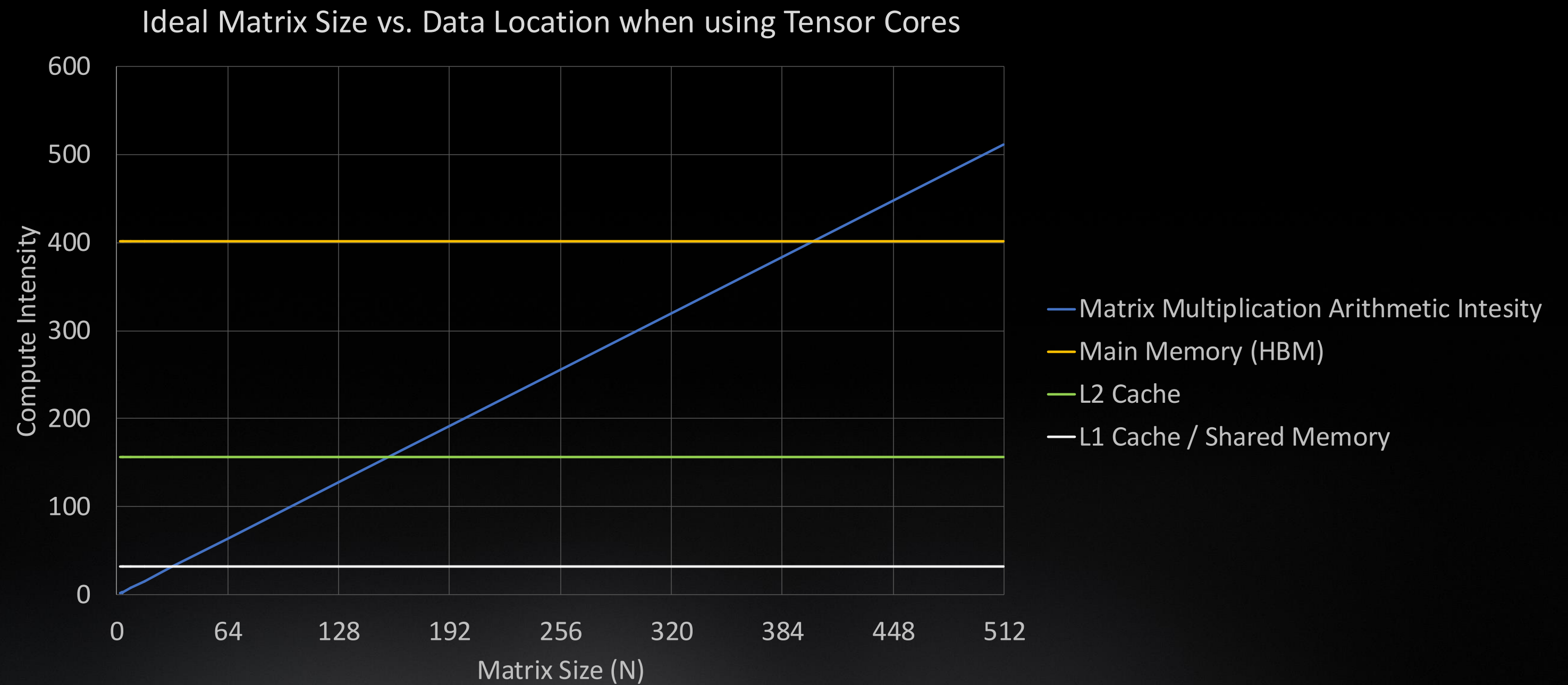
# WORKING WITH SMALLER MATRICES

B/W    Latency

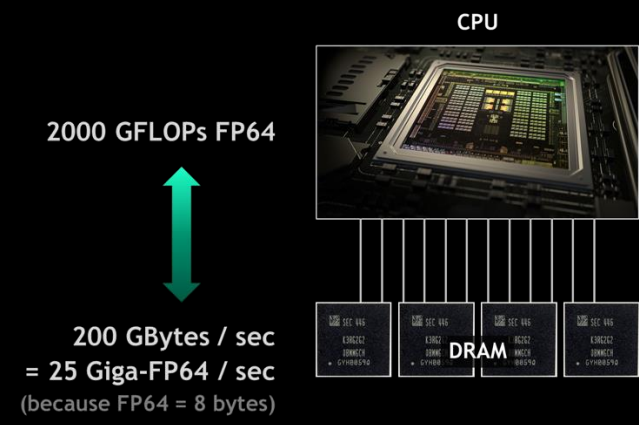


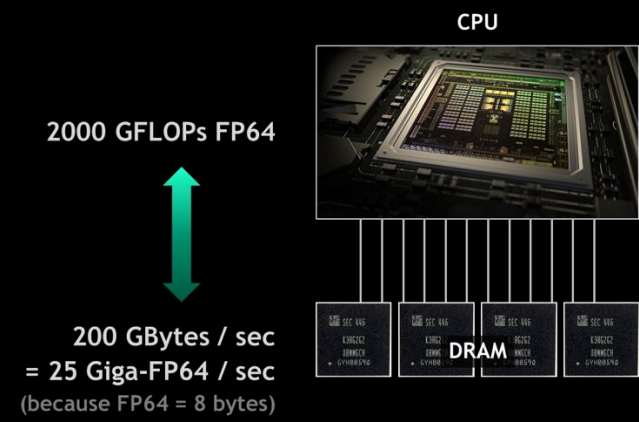
Data Location	Bandwidth (GB/sec)	Compute Intensity	Tensor Core Compute Intensity
L1 / Shared	19,400	8	32
L2 Cache	4,000	39	156
HBM	1,555	100	401
NVLink	300	520	2080
PCIe	25	6240	24960

# WORKING WITH SMALLER MATRICES

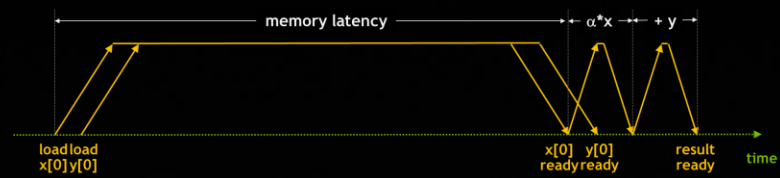


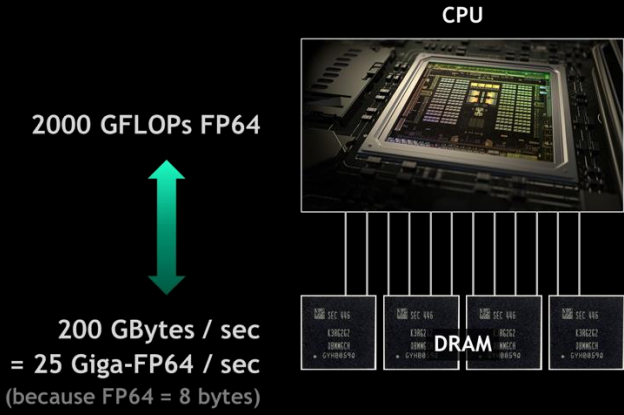




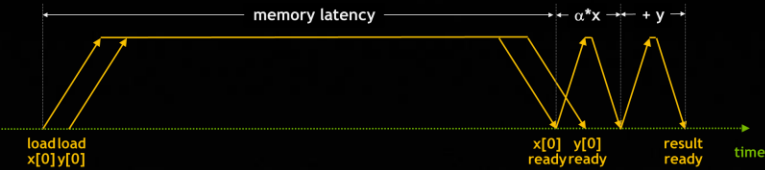


```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```



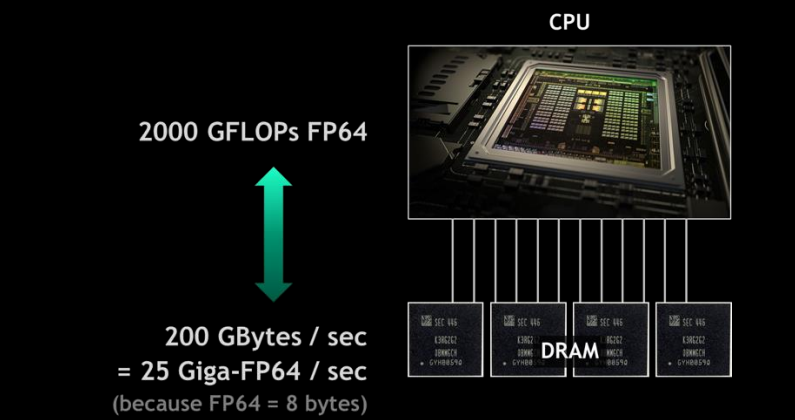


```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

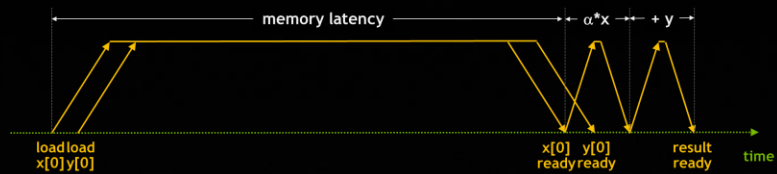


	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x

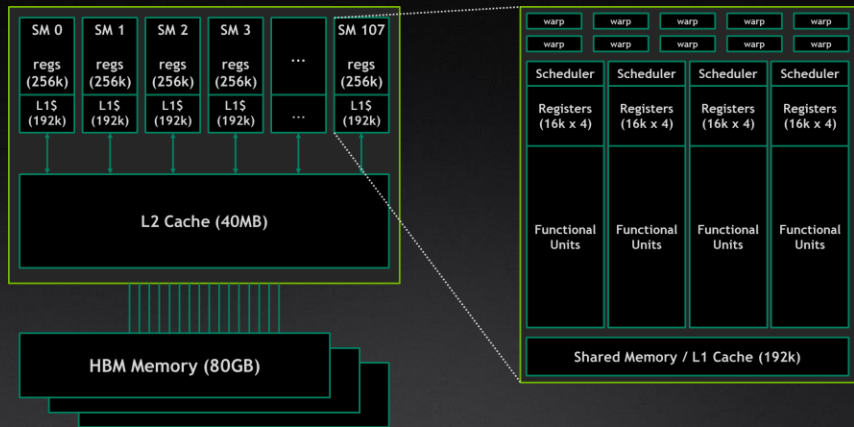


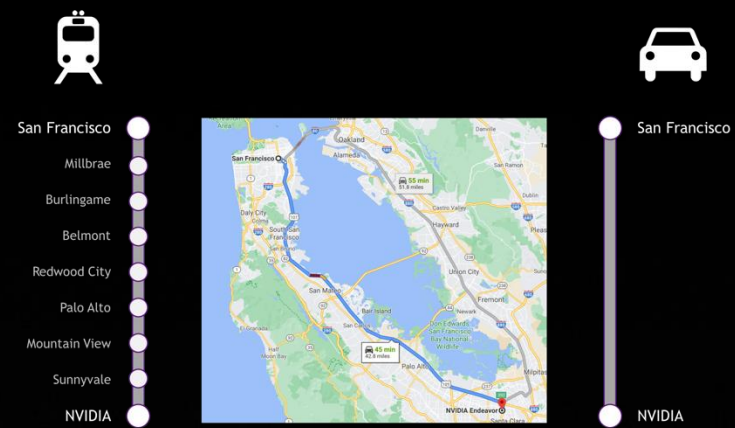
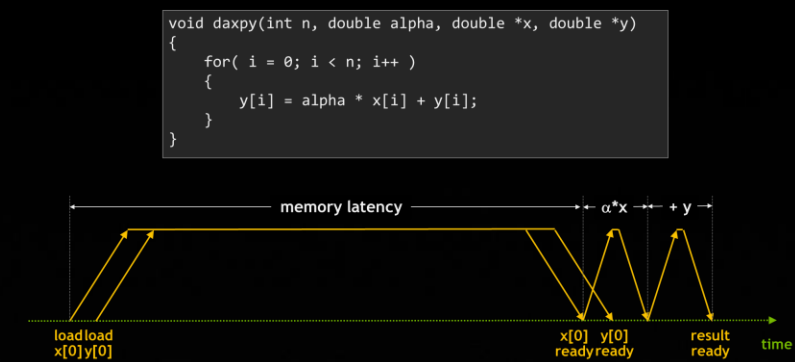
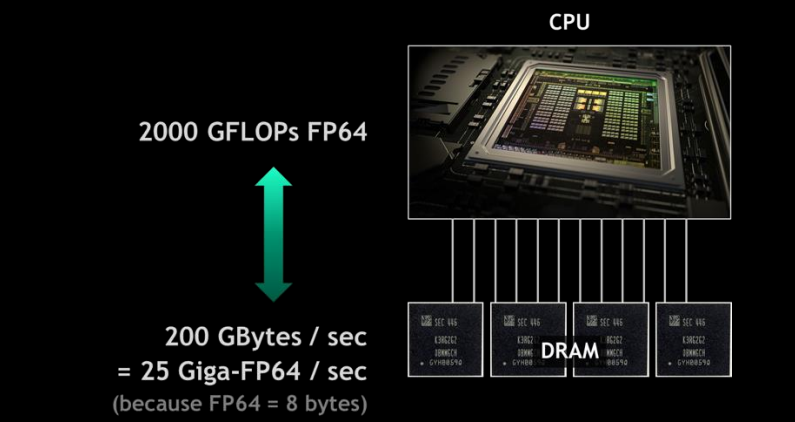


```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

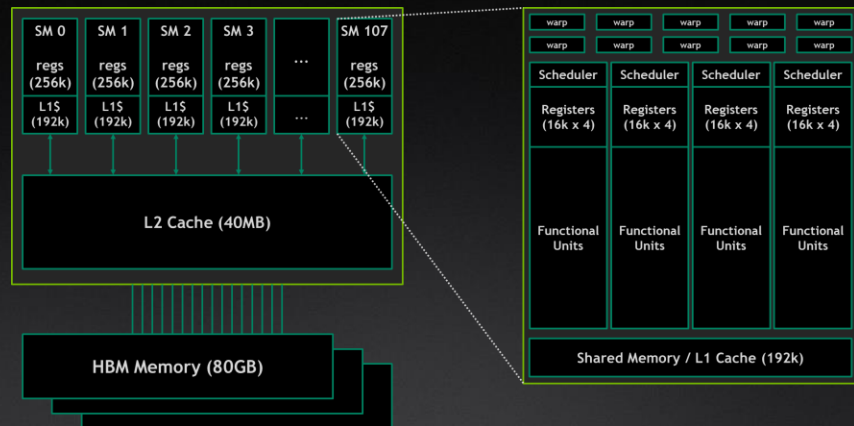


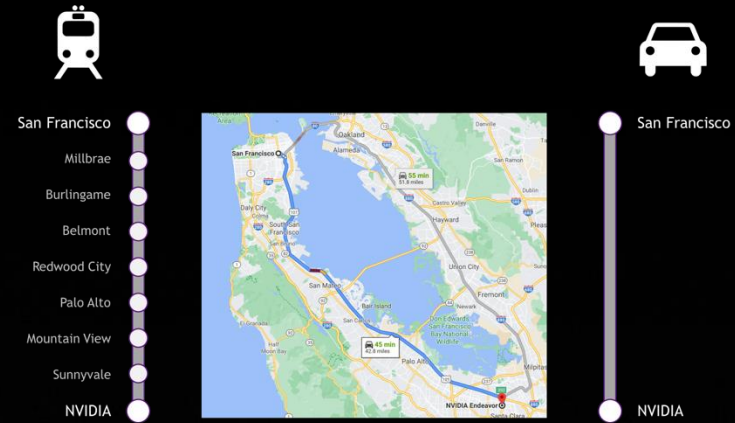
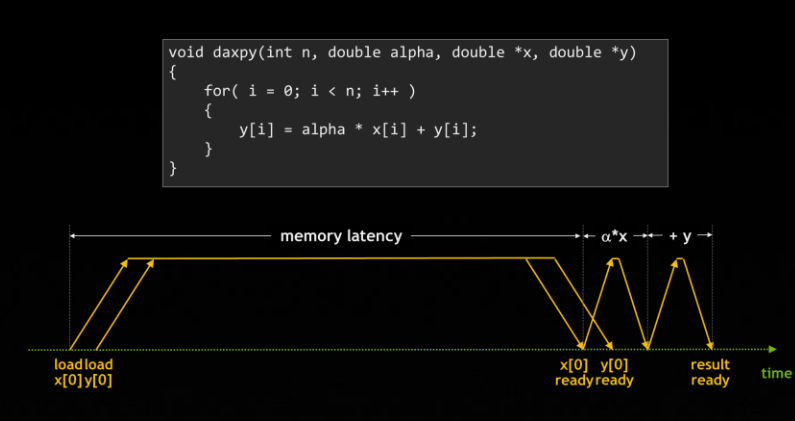
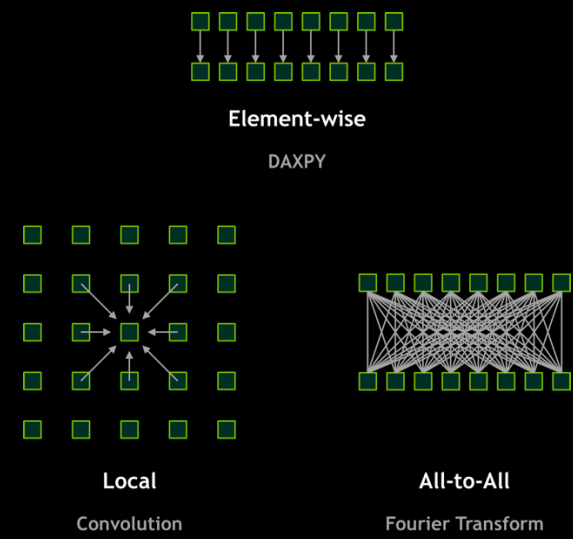
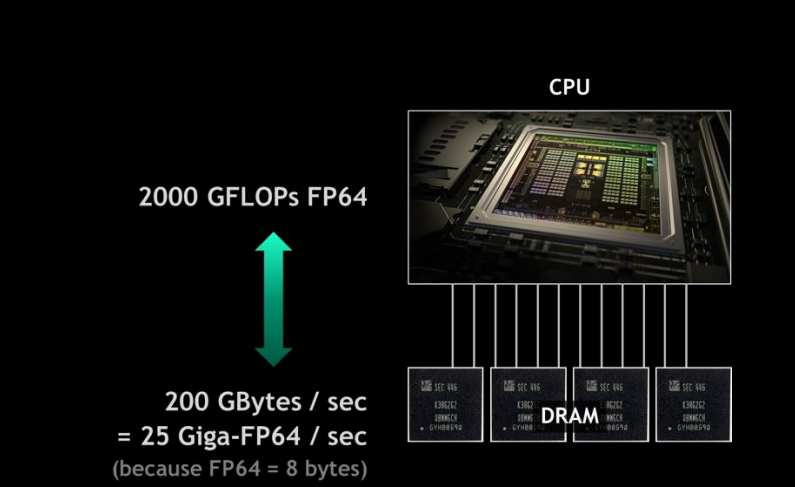
	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x



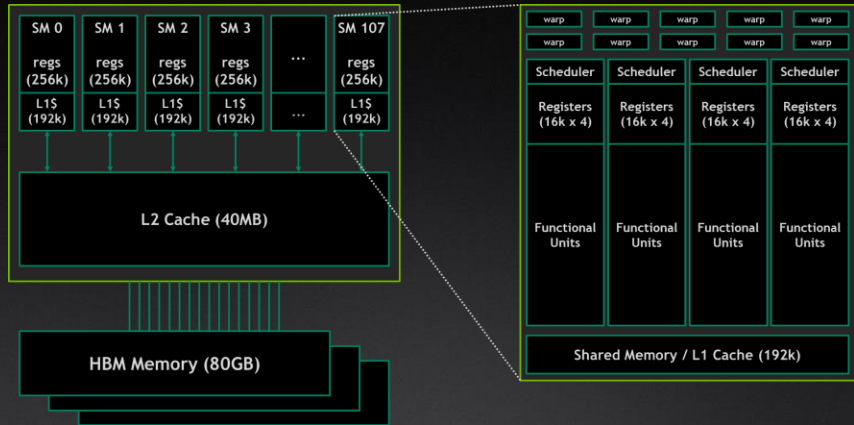


	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x





	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x

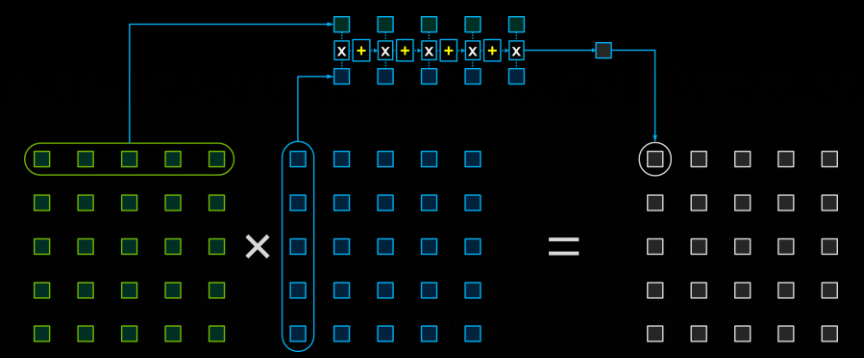
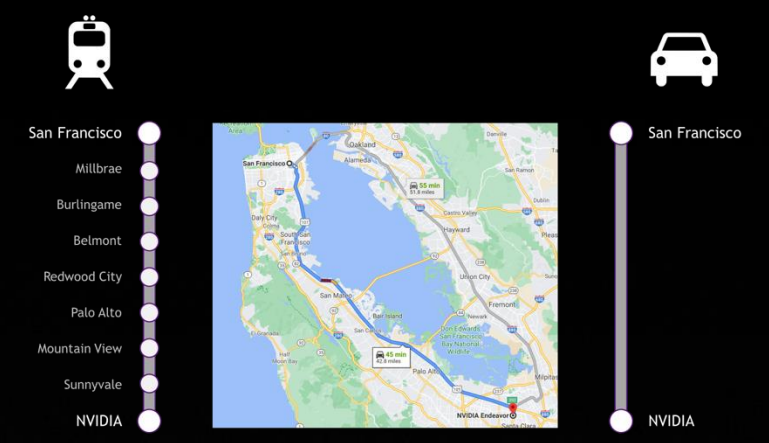
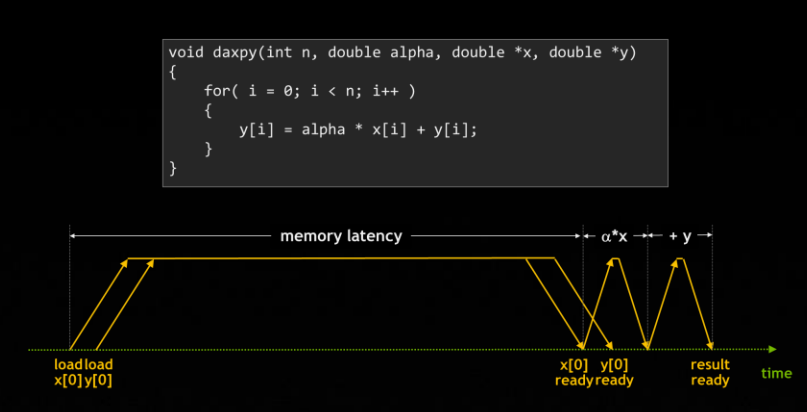
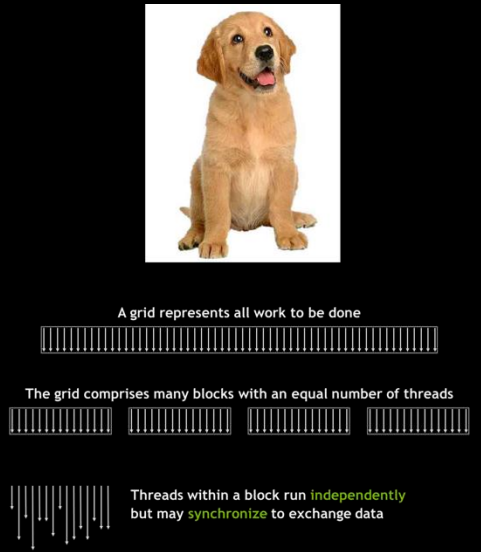
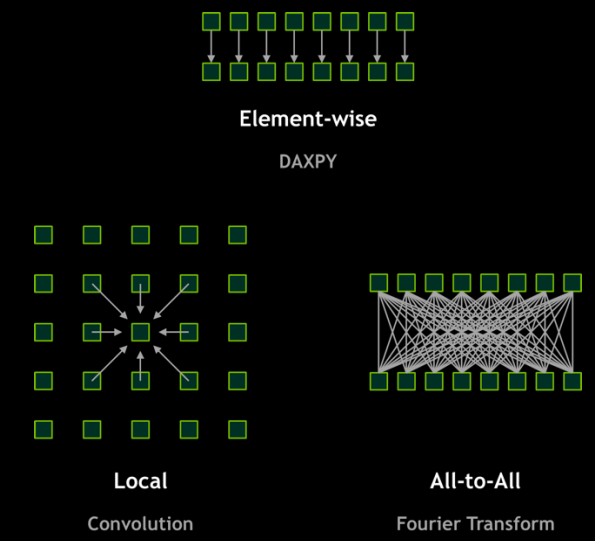
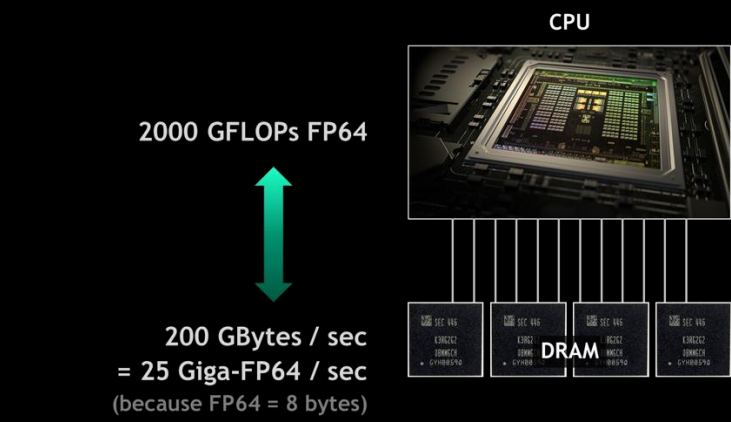




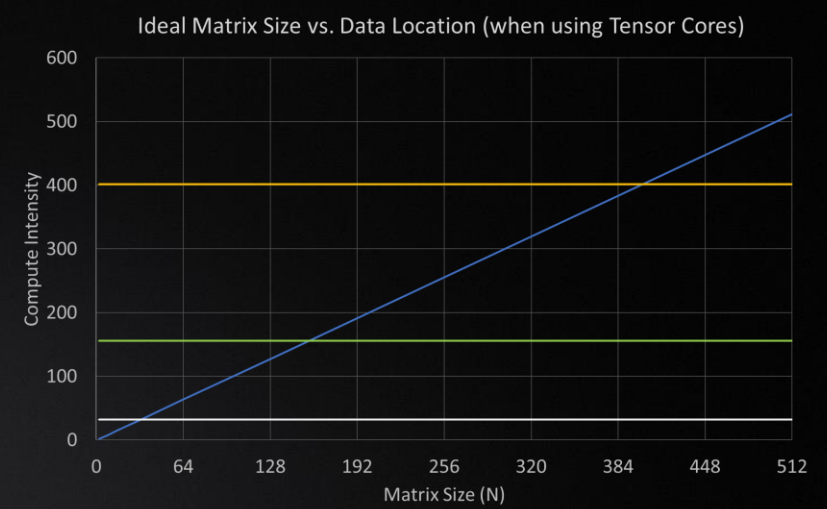
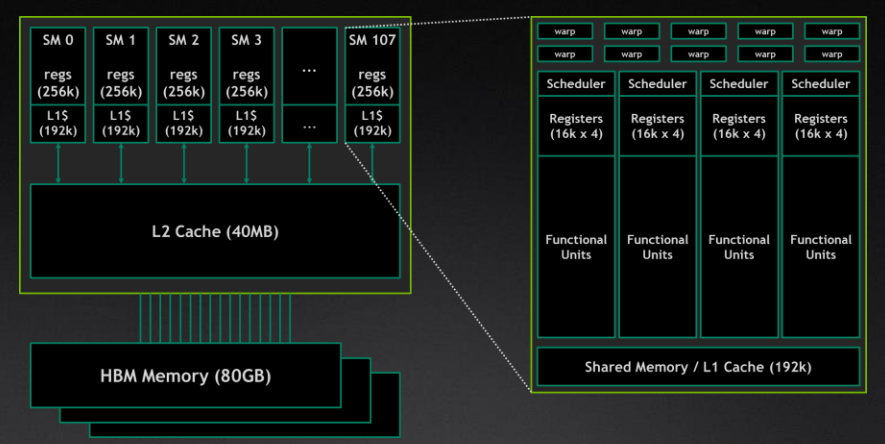








	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	143
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	12,727
Memory efficiency	0.0025%	0.064%	0.13%
Threads required	39,264	1,556	729
Threads available	221,184	2048	896
Thread ratio	5.6x	1.3x	1.2x







WHERE'S MY DATA?

~~WHY~~  
HOW GPU COMPUTING WORKS

Stephen Jones, GTC 2021

