CUDA OPTIMIZATION TIPS, TRICKS AND TECHNIQUES

Stephen Jones, GTC 2017

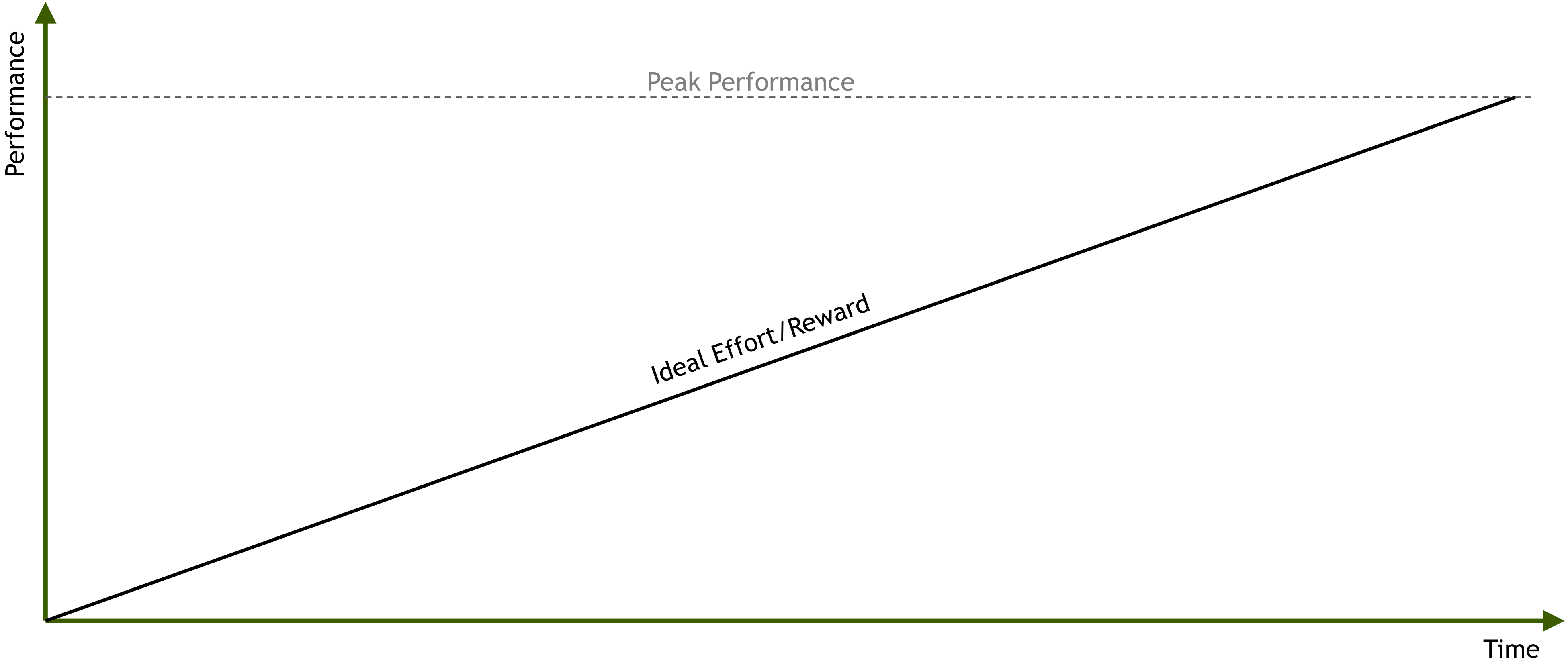
optimization

noun | op·ti·mi·za·tion | \ˌäp-tə-mə-ˈzā-shən\

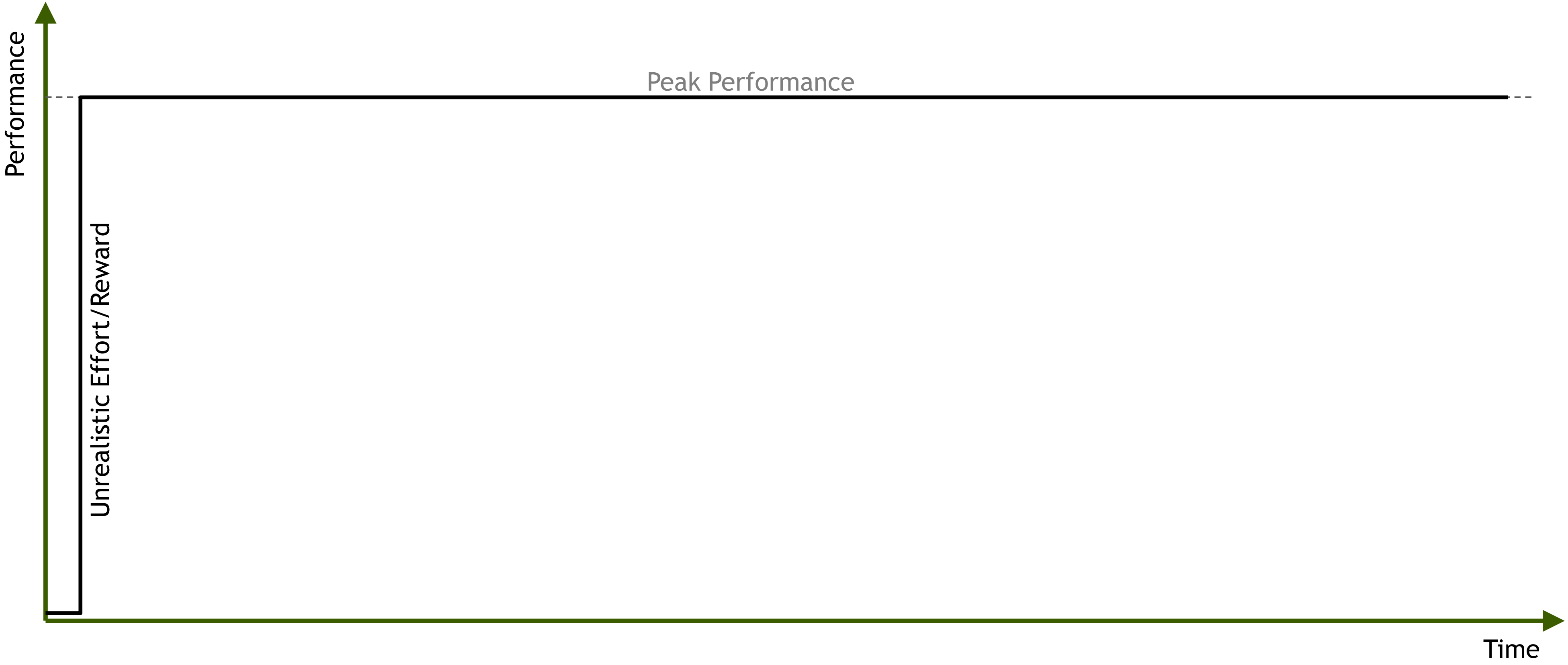
Popularity: Bottom 50% of words

The art of doing **more** with **less**

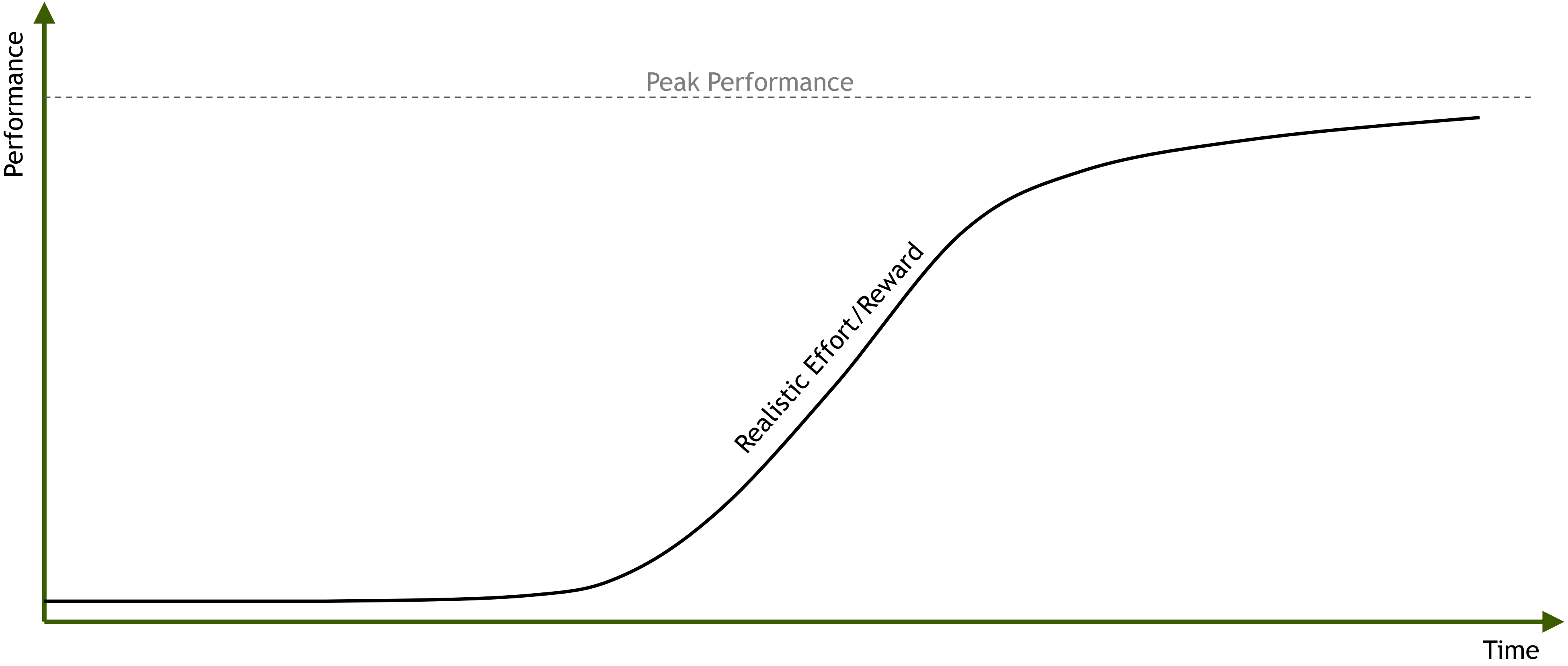
RULE #1: DON’T TRY TOO HARD



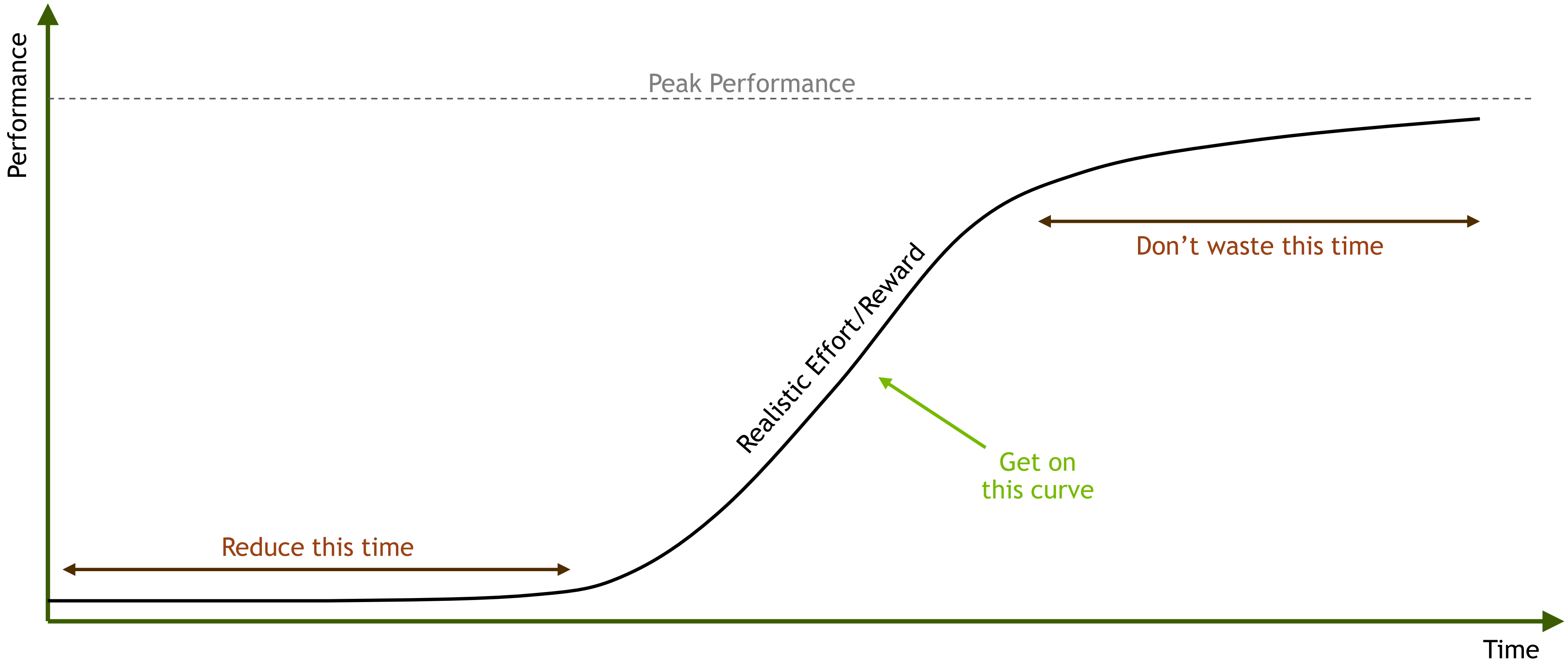
RULE #1: DON'T TRY TOO HARD



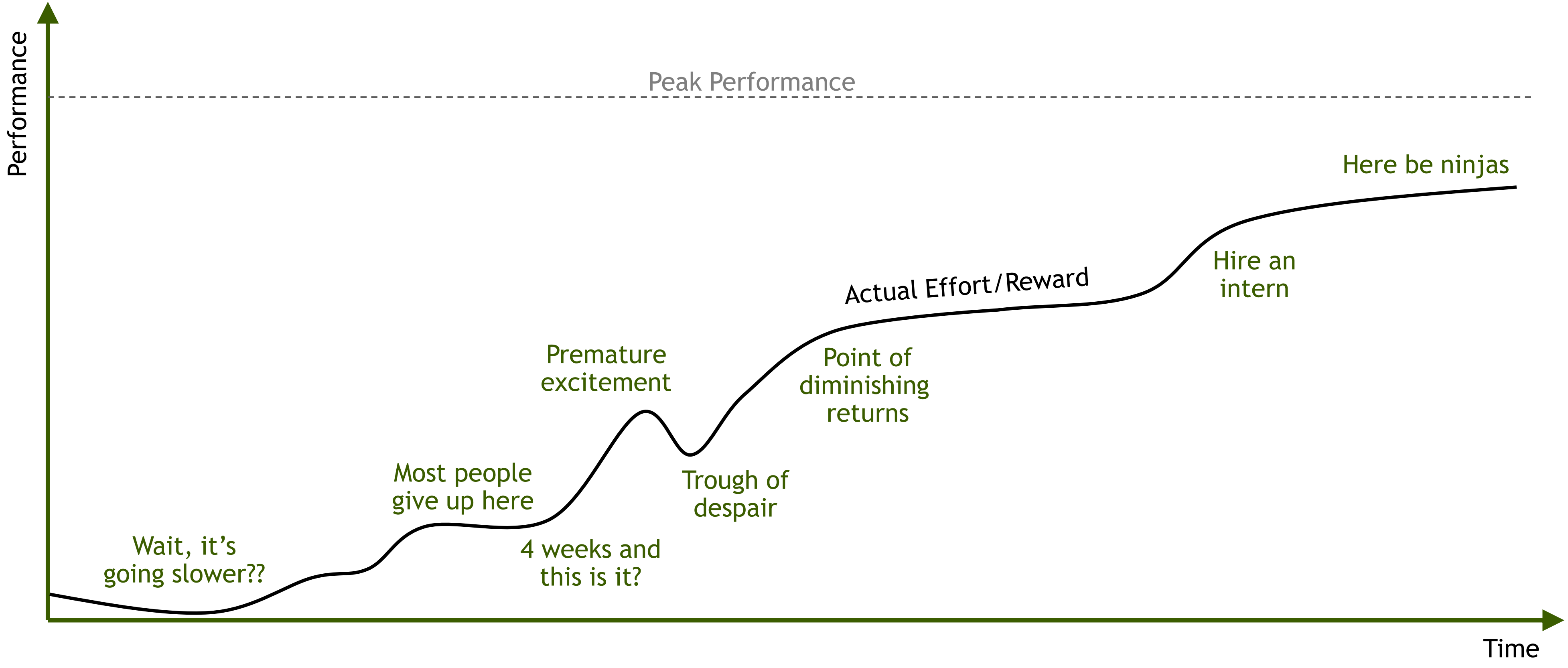
RULE #1: DON'T TRY TOO HARD



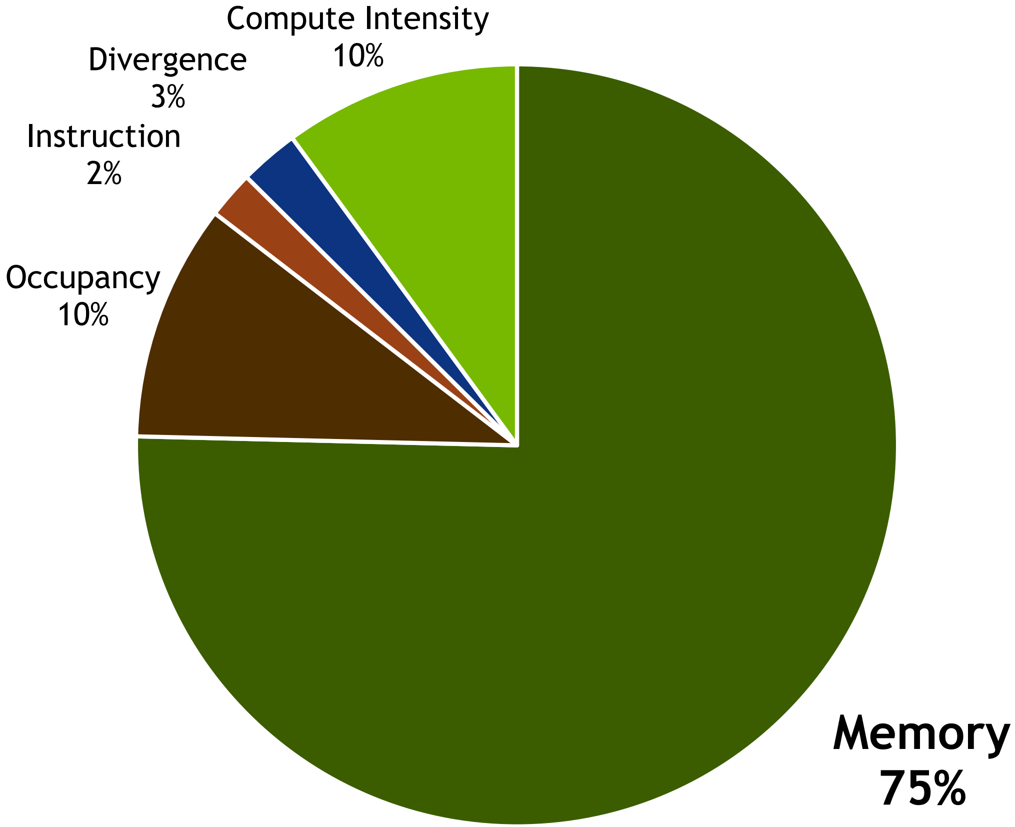
RULE #1: DON’T TRY TOO HARD



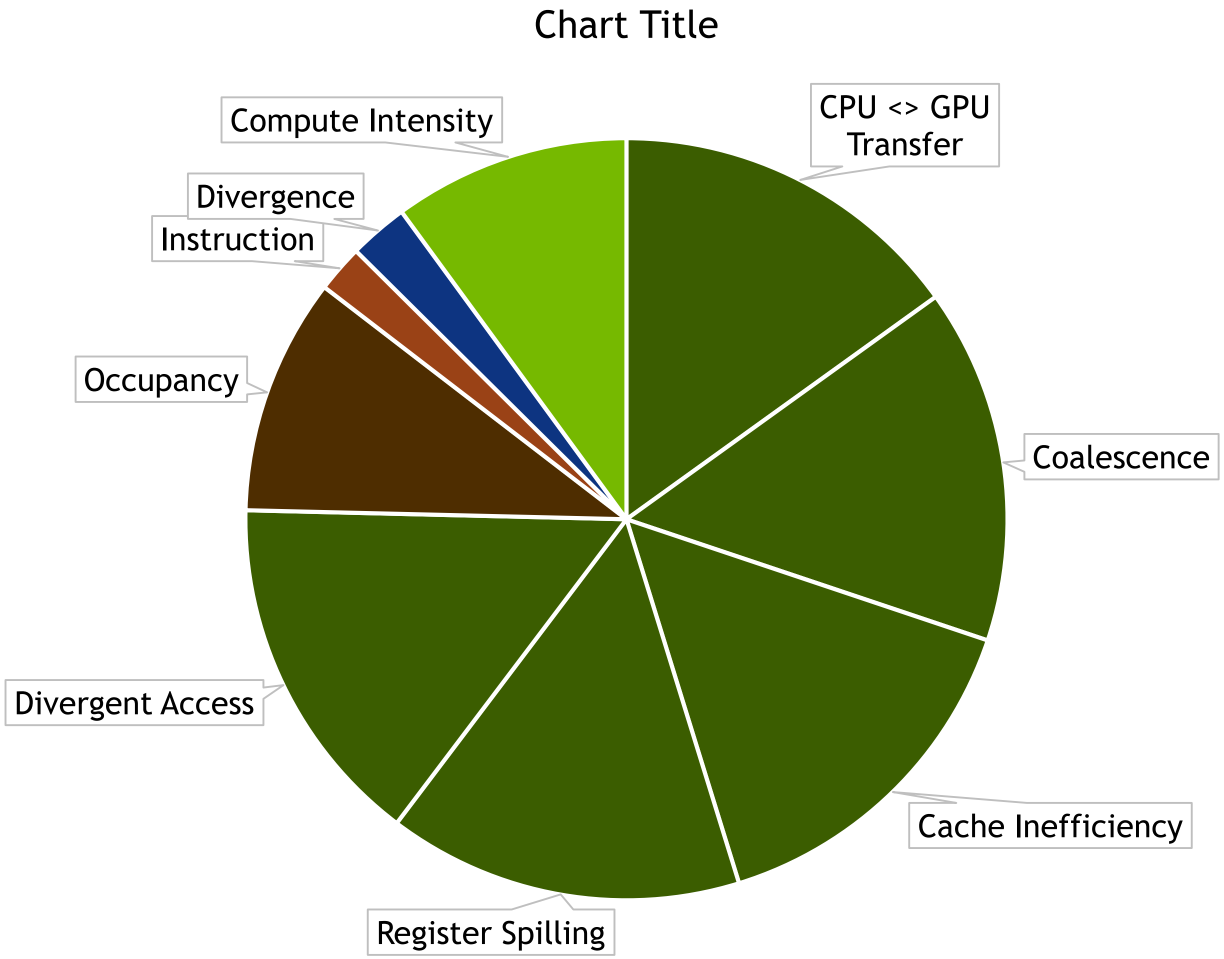
RULE #1: DON’T TRY TOO HARD



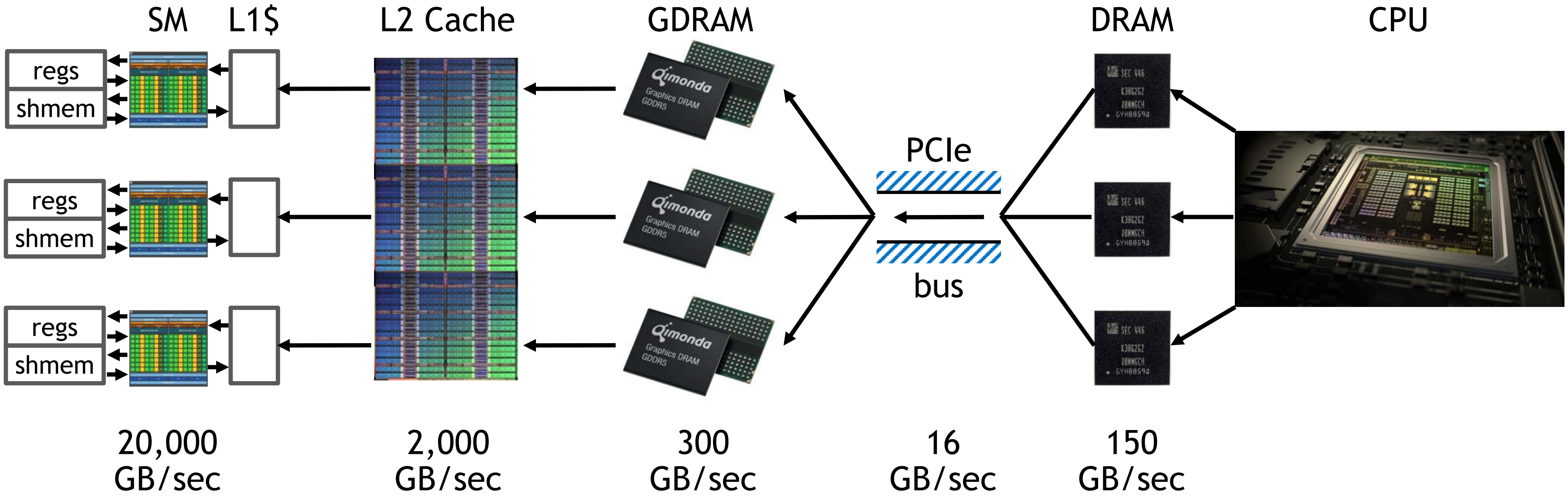
PERFORMANCE CONSTRAINTS



PERFORMANCE CONSTRAINTS



MEMORY ORDERS OF MAGNITUDE



SM

L1$

L2 Cache

GDRAM

DRAM

CPU

regs

shmem

regs

shmem

regs

shmem

PCIe

bus

20,000
GB/sec

2,000
GB/sec

300
GB/sec

16
GB/sec

150
GB/sec

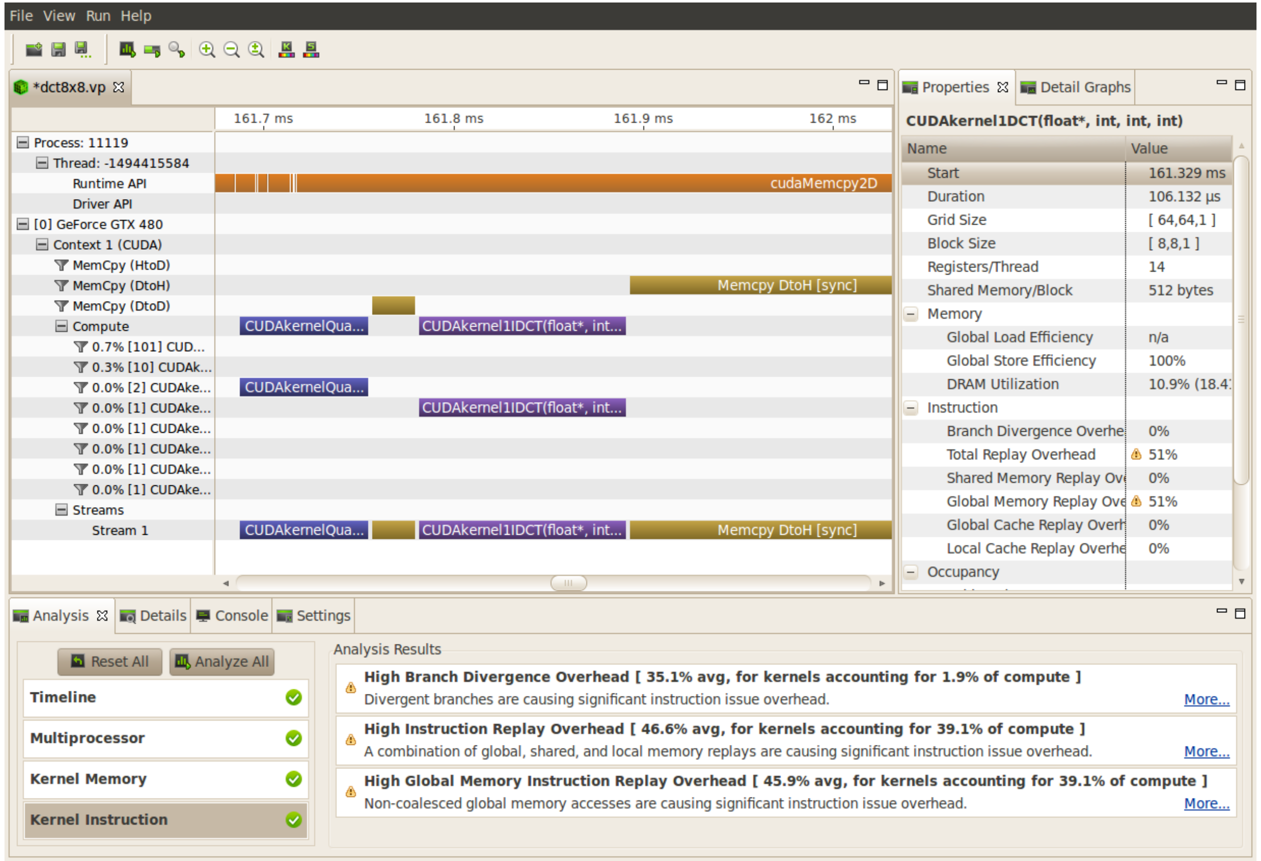
TALK BREAKDOWN

In no particular order

1. Why Didn’t I Think Of That?
2. CPU Memory to GPU Memory (the PCIe Bus)
3. GPU Memory to the SM
4. Registers & Shared Memory
5. Occupancy, Divergence & Latency
6. Weird Things You Never Thought Of (and probably shouldn’t try)

WHERE TO BEGIN?

THE OBVIOUS



Process: 11119
Thread: -1494415584
Runtime API
Driver API
[0] GeForce GTX 480
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
0.7% [101] CUD...
0.3% [10] CUDAk...
0.0% [2] CUDAke...
0.0% [1] CUDAke...
0.0% [1] CUDAke...
0.0% [1] CUDAke...
0.0% [1] CUDAke...
0.0% [1] CUDAke...
Streams
Stream 1

CUDAkernel1DCT(float*, int, int, int)

Name	Value
Start	161.329 ms
Duration	106.132 µs
Grid Size	[64,64,1]
Block Size	[8,8,1]
Registers/Thread	14
Shared Memory/Block	512 bytes
Memory	
Global Load Efficiency	n/a
Global Store Efficiency	100%
DRAM Utilization	10.9% (18.4
Instruction	
Branch Divergence Overhe	0%
Total Replay Overhead	⚠ 51%
Shared Memory Replay Ov	0%
Global Memory Replay Ove	⚠ 51%
Global Cache Replay Overh	0%
Local Cache Replay Overhe	0%
Occupancy	

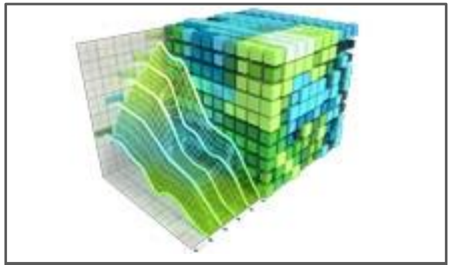
- Timeline ✓
- Multiprocessor ✓
- Kernel Memory ✓
- Kernel Instruction ✓

Analysis Results

⚠ **High Branch Divergence Overhead [35.1% avg, for kernels accounting for 1.9% of compute]**
Divergent branches are causing significant instruction issue overhead. More...

⚠ **High Instruction Replay Overhead [46.6% avg, for kernels accounting for 39.1% of compute]**
A combination of global, shared, and local memory replays are causing significant instruction issue overhead. More...

⚠ **High Global Memory Instruction Replay Overhead [45.9% avg, for kernels accounting for 39.1% of compute]**
Non-coalesced global memory accesses are causing significant instruction issue overhead. More...

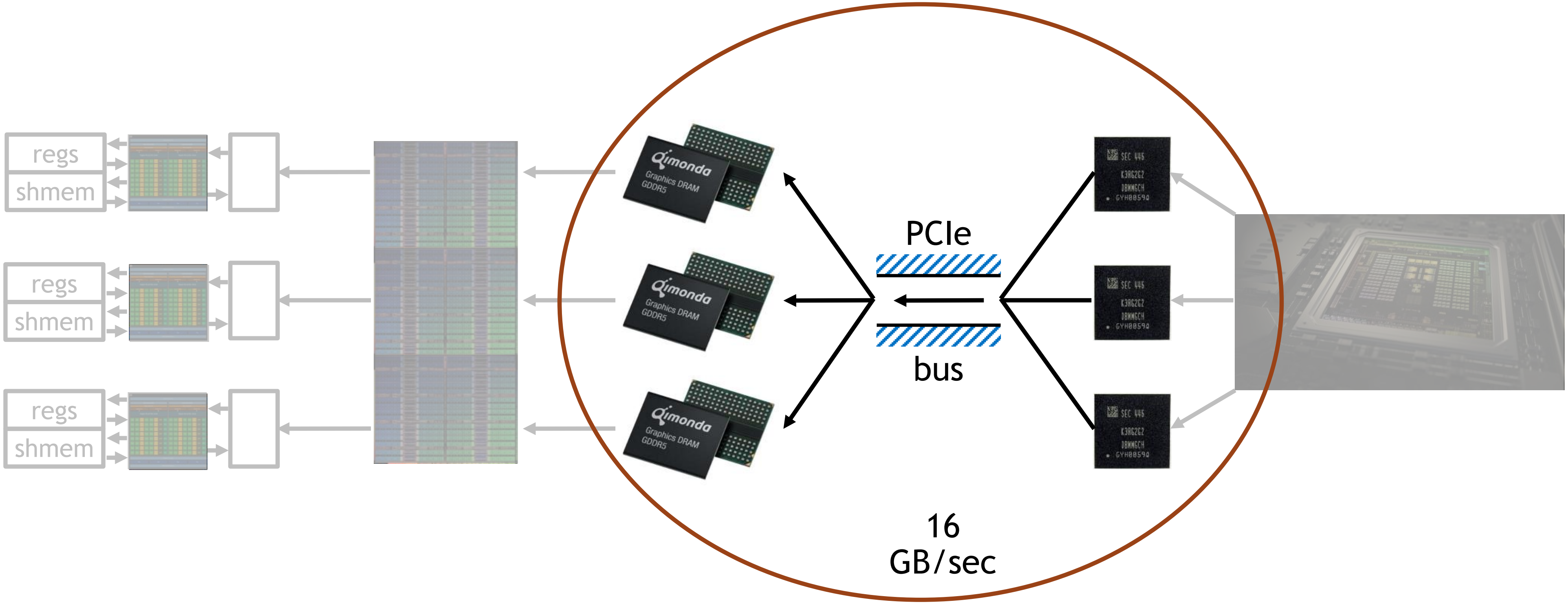


NVIDIA Visual Profiler

Start with the Visual Profiler

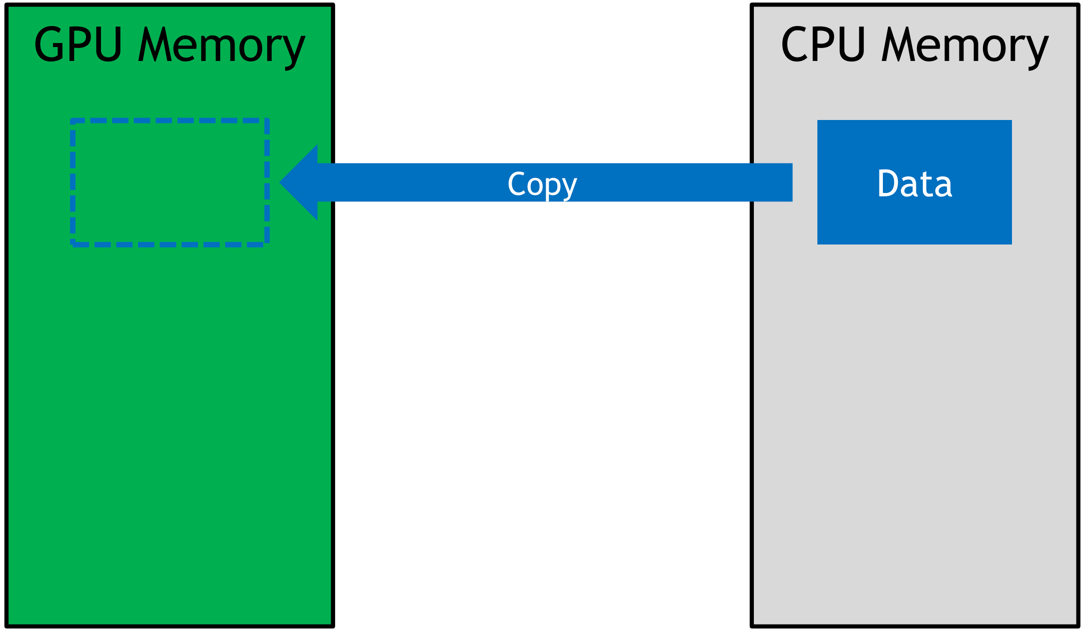
CPU <> GPU DATA MOVEMENT

PCI ISSUES

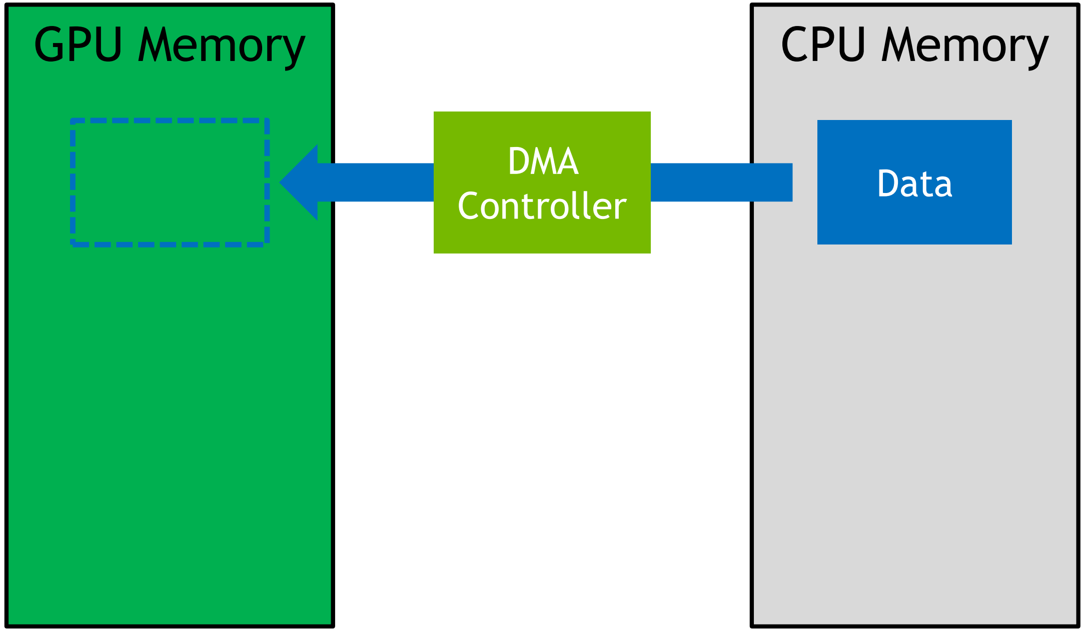


Moving data over the PCIe bus

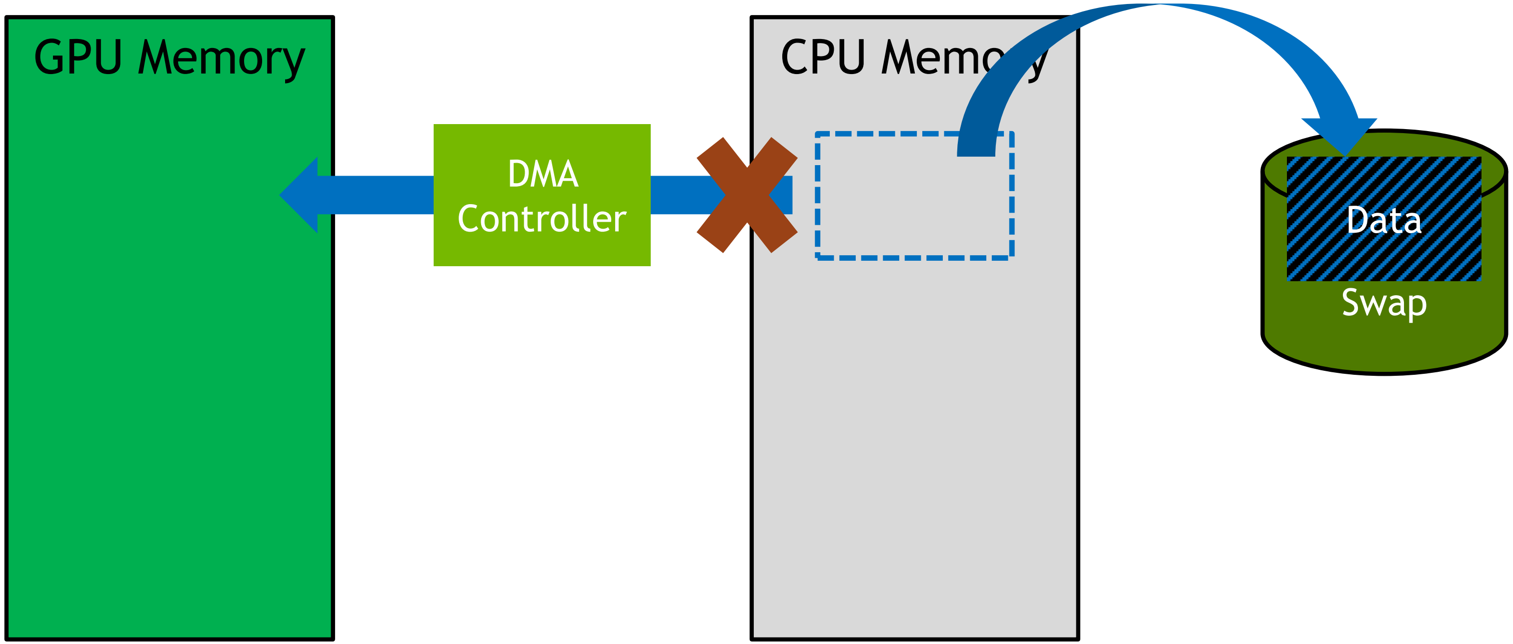
PIN YOUR CPU MEMORY



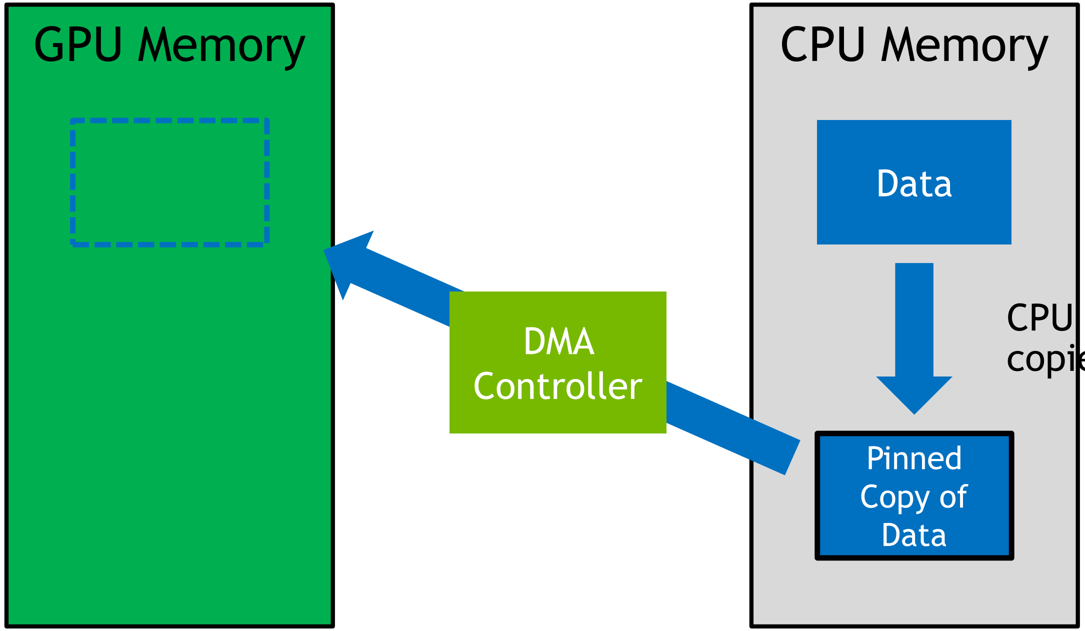
PIN YOUR CPU MEMORY



PIN YOUR CPU MEMORY

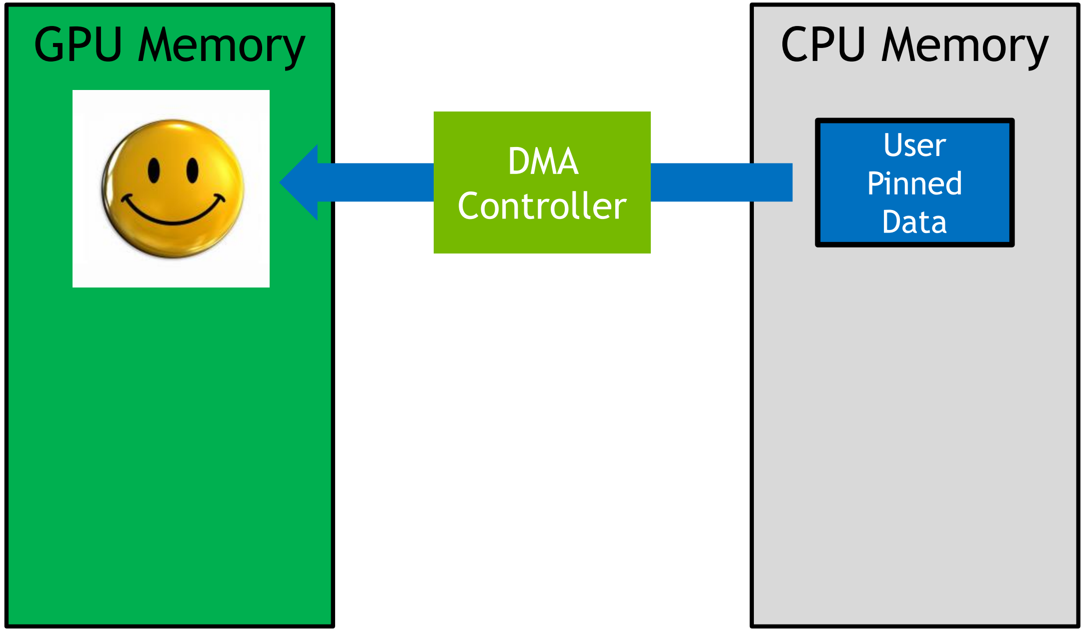


PIN YOUR CPU MEMORY



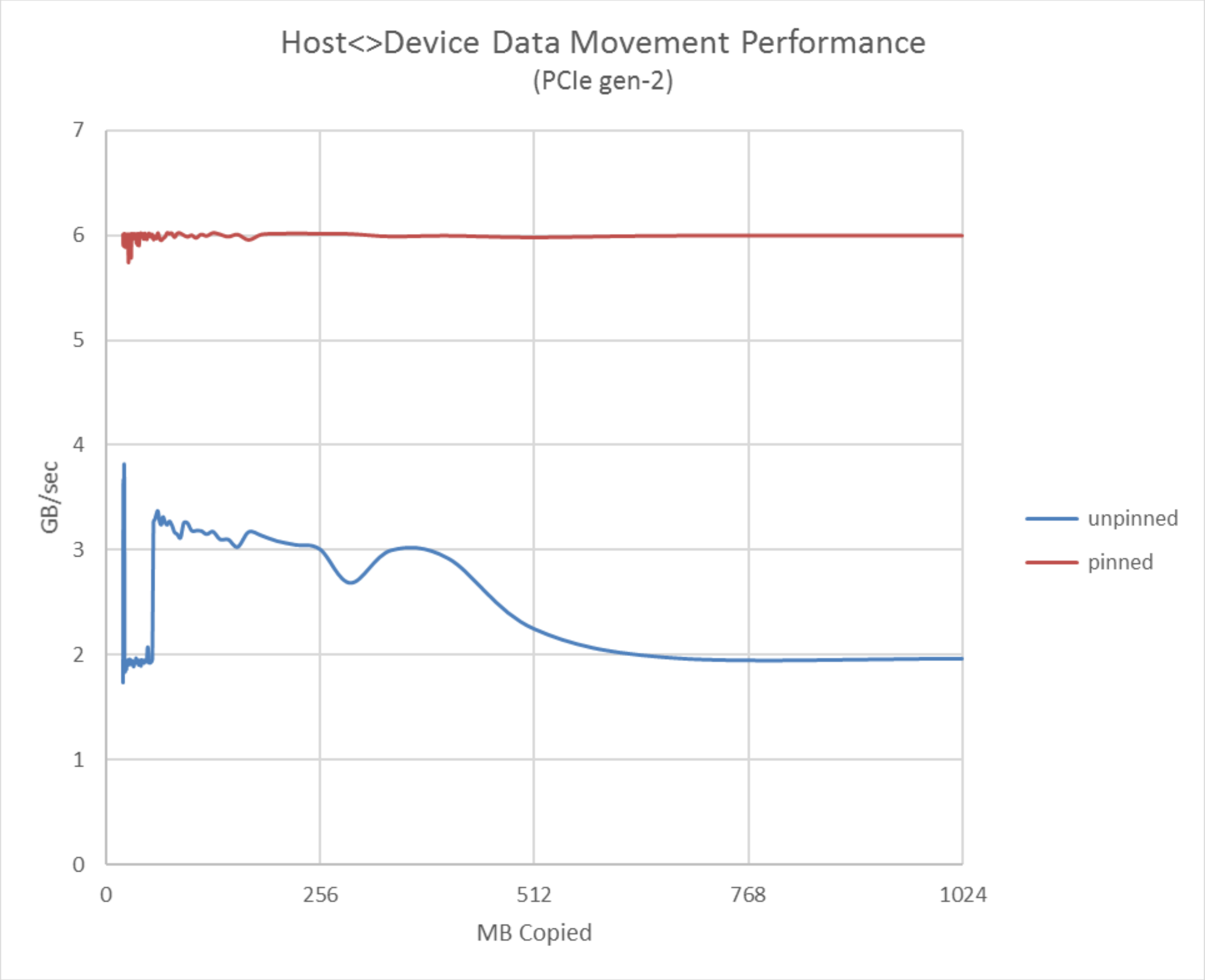
CPU allocates & pins page then copies locally before DMA

PIN YOUR CPU MEMORY

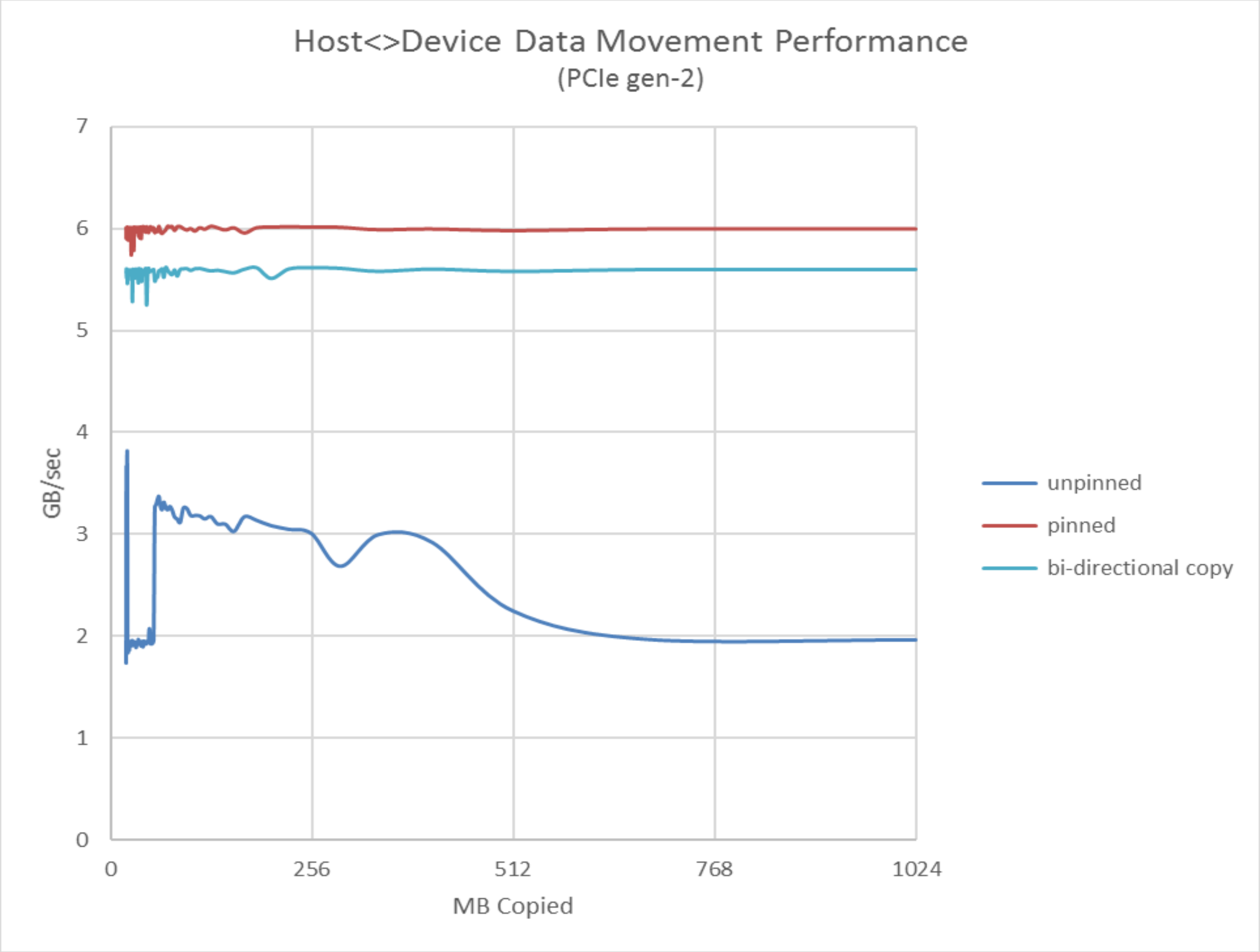


```
  cudaHostAlloc( &data, size, cudaHostAllocMapped );
cudaHostRegister( &data, size, cudaHostRegisterDefault );
```

PIN YOUR CPU MEMORY



REMEMBER: PCIe GOES BOTH WAYS

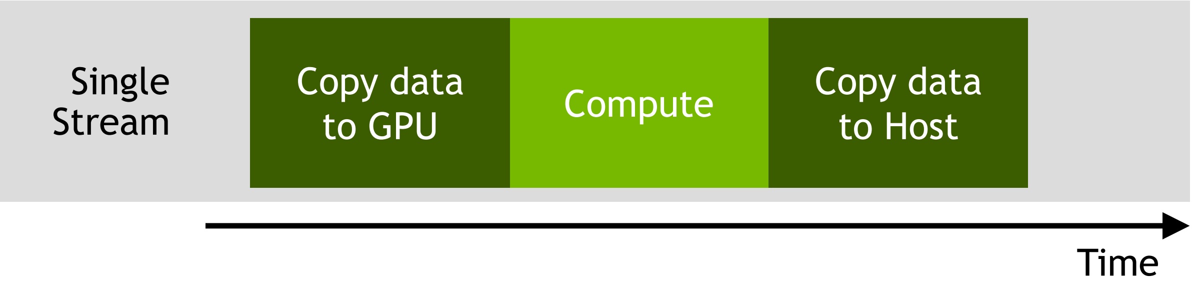


STREAMS & CONCURRENCY

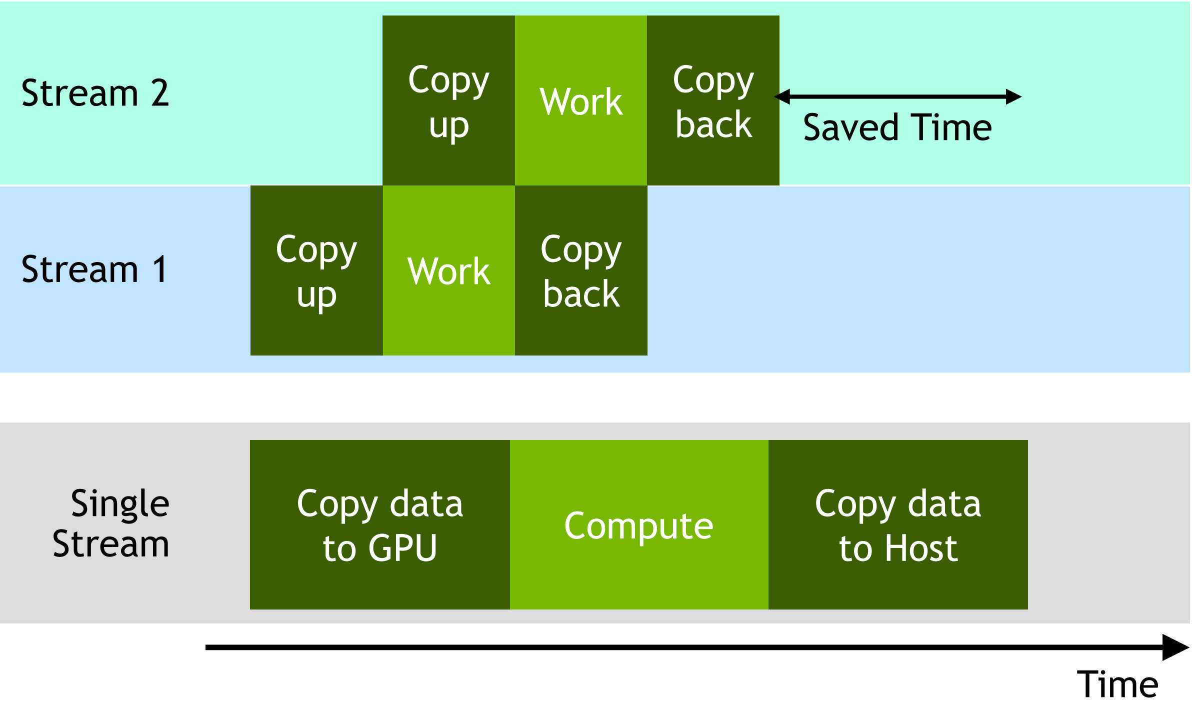
Hiding the cost of data transfer

Operations in a single stream are ordered

But hardware can copy and compute at the same time

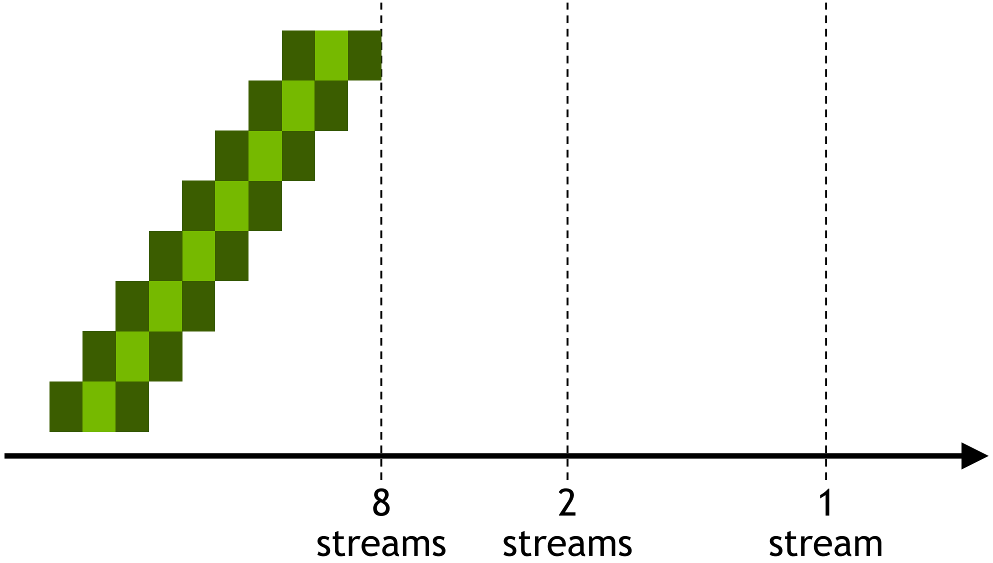


STREAMS & CONCURRENCY



STREAMS & CONCURRENCY

Can keep on breaking work into smaller chunks and saving time

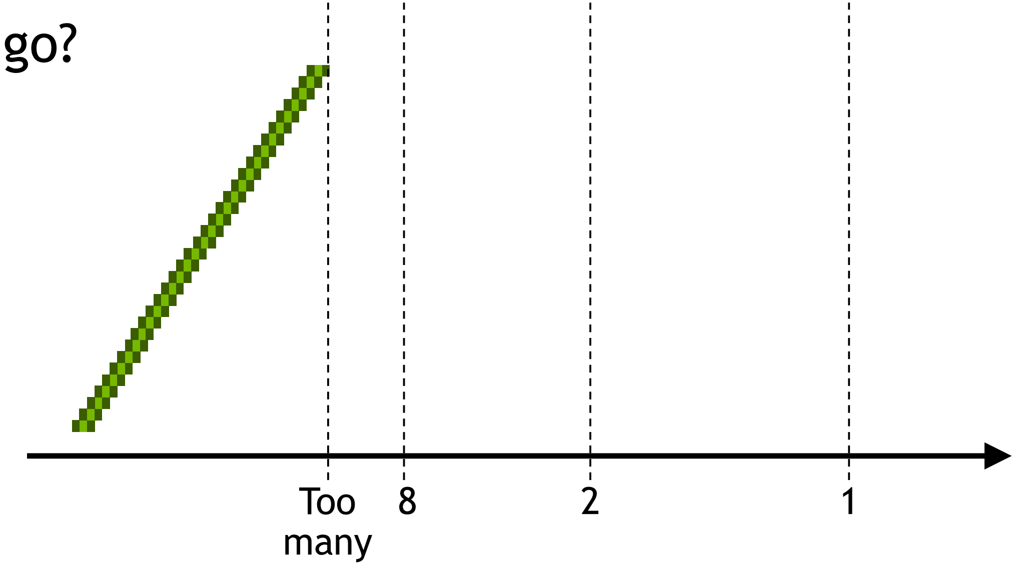


SMALL PCIe TRANSFERS

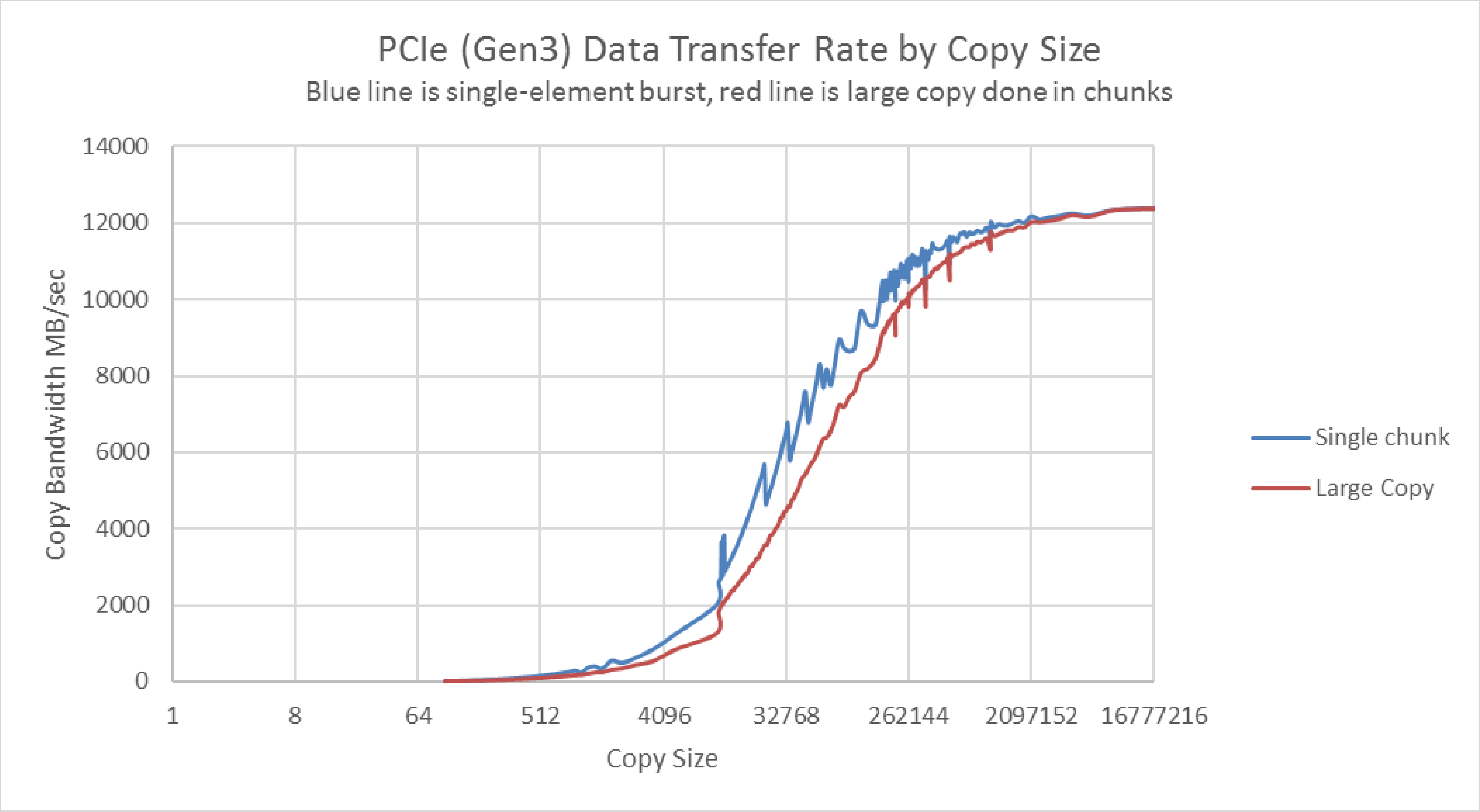
PCIe is designed for large data transfers

But fine-grained copy/compute overlap prefers small transfers

So how small can we go?

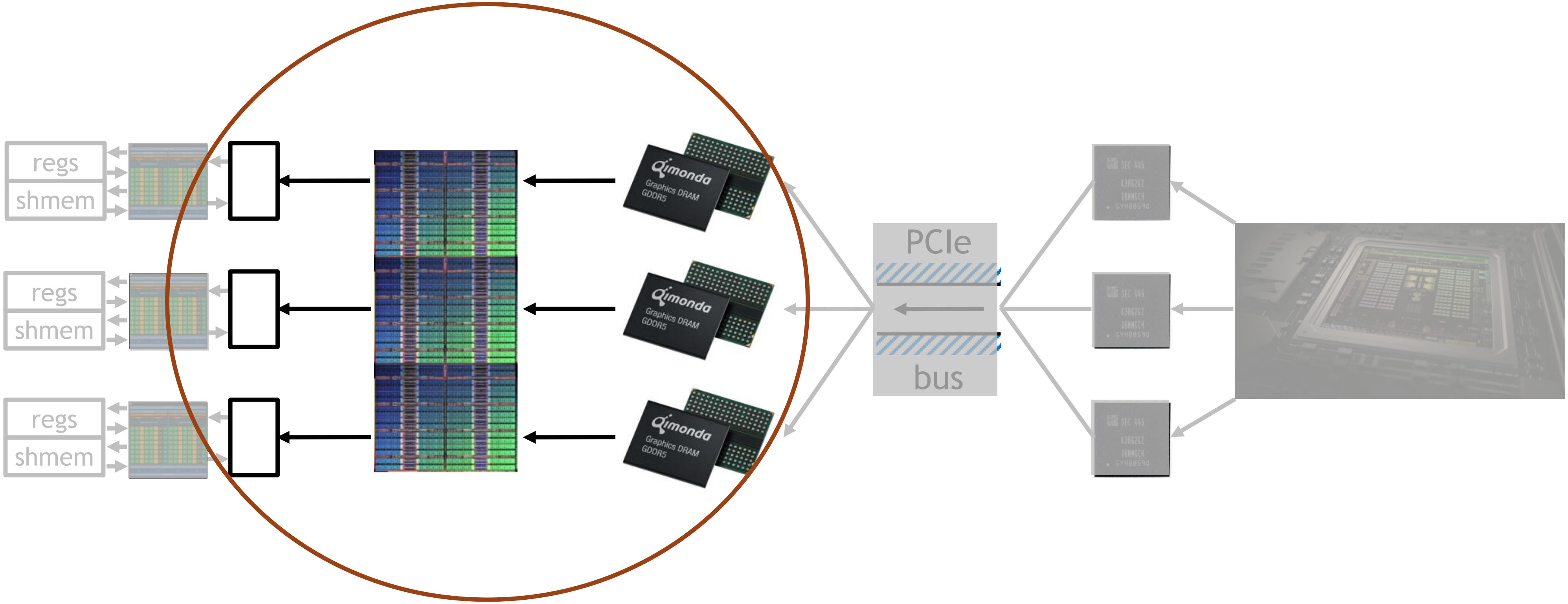


APPARENTLY NOT THAT SMALL



FROM GPU MEMORY TO GPU THREADS

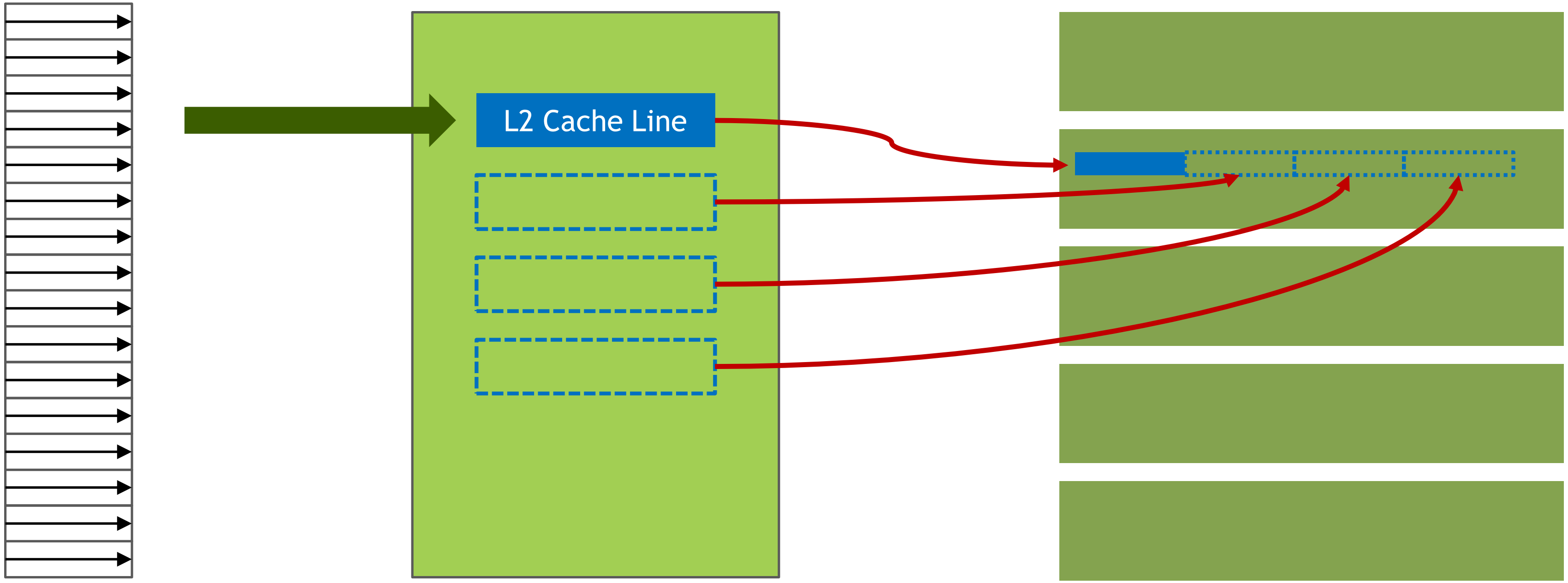
FEEDING THE MACHINE



From GPU Memory to the SMs

USE THE PARALLEL ARCHITECTURE

Hardware is optimized to use all SIMT threads at once



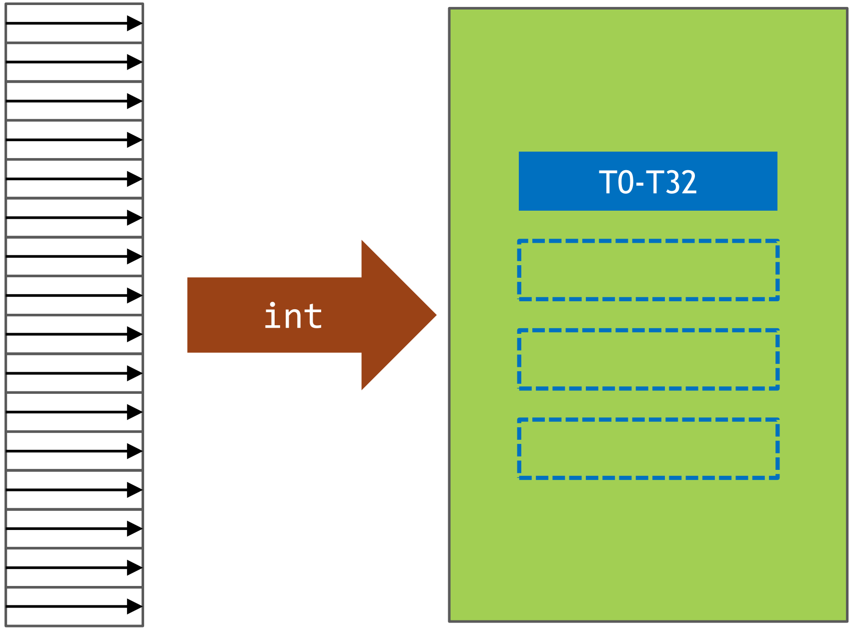
Threads run
in groups of 32

Cache is sized to service
sets of 32 requests at a time

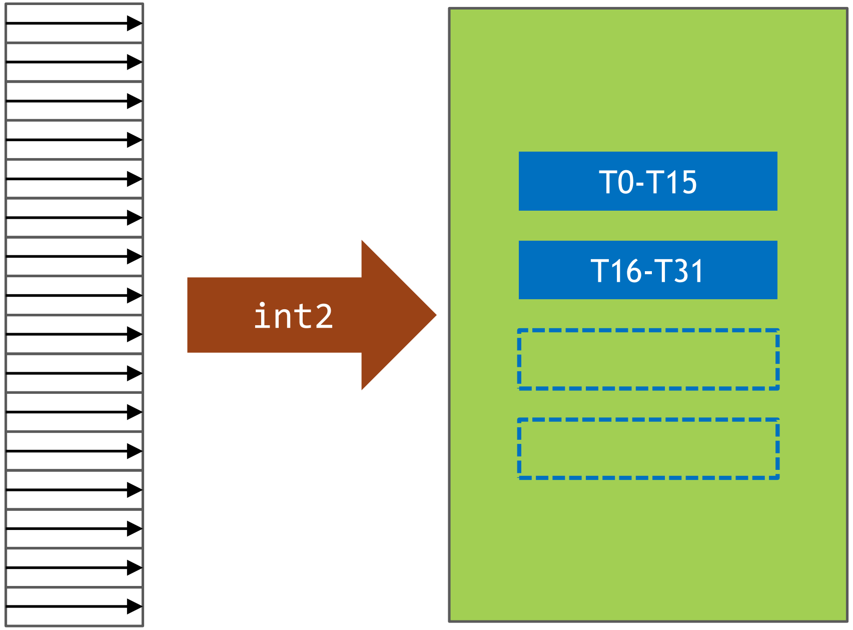
High-speed GPU memory
works best with linear access

VECTORIZE MEMORY LOADS

Multi-Word as well as Multi-Thread

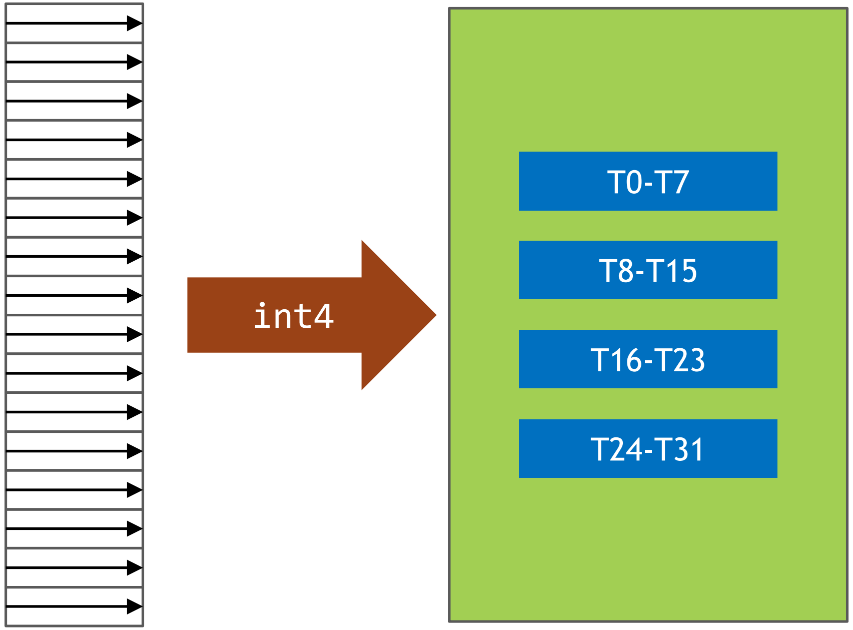


VECTORIZE MEMORY LOADS



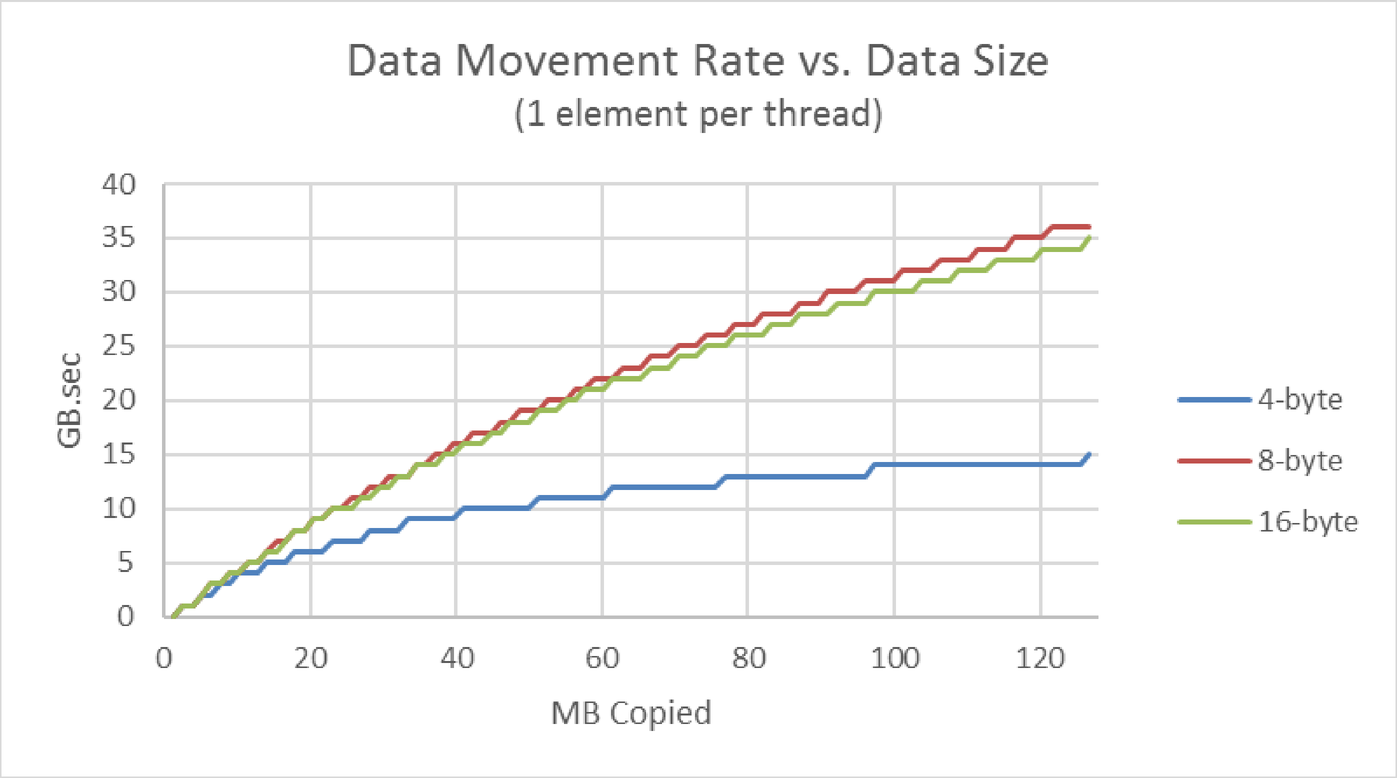
Fill multiple cache lines in a single fetch

VECTORIZE MEMORY LOADS



Fill multiple cache lines in a single fetch

VECTORIZE MEMORY LOADS



DO MULTIPLE LOADS PER THREAD

Multi-Thread, Multi-Word **AND** Multi-Iteration

```
__global__ void copy(int2 *input,
                     int2 *output,
                     int max) {

  int id = threadIdx.x +
           blockDim.x * blockIdx.x;

  if( id < max ) {
    output[id] = input[id];
  }
}
```

One copy per thread
Maximum overhead

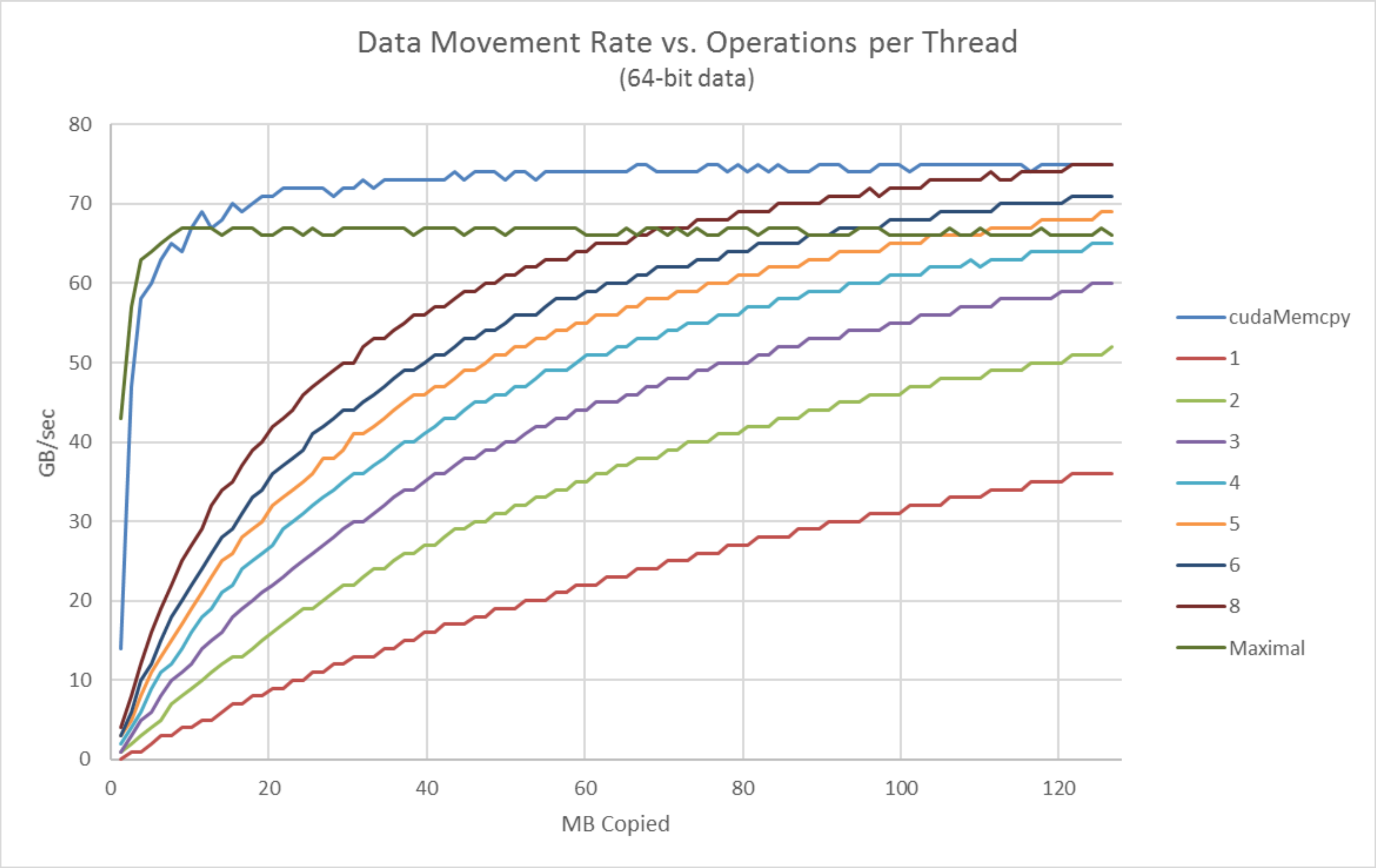
```
__global__ void copy(int2 *input,
                     int2 *output,
                     int max,
                     int loadsPerThread) {

  int id = threadIdx.x +
           blockDim.x * blockIdx.x;

  for(int n=0; n<loadsPerThread; n++) {
    if( id >= max ) {
      break;
    }
    output[id] = input[id];
    id += blockDim.x * gridDim.x;
  }
}
```

Multiple copies per thread
Amortize overhead

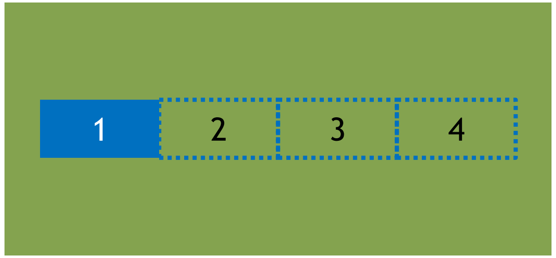
“MAXIMAL” LAUNCHES ARE BEST



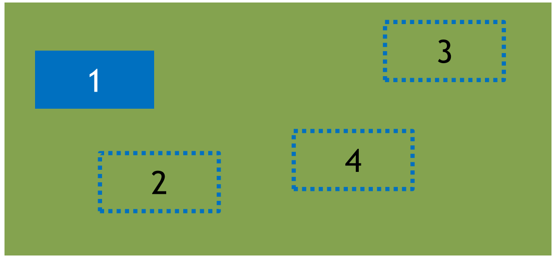
COALESCED MEMORY ACCESS

It’s not just good enough to use all SIMT threads

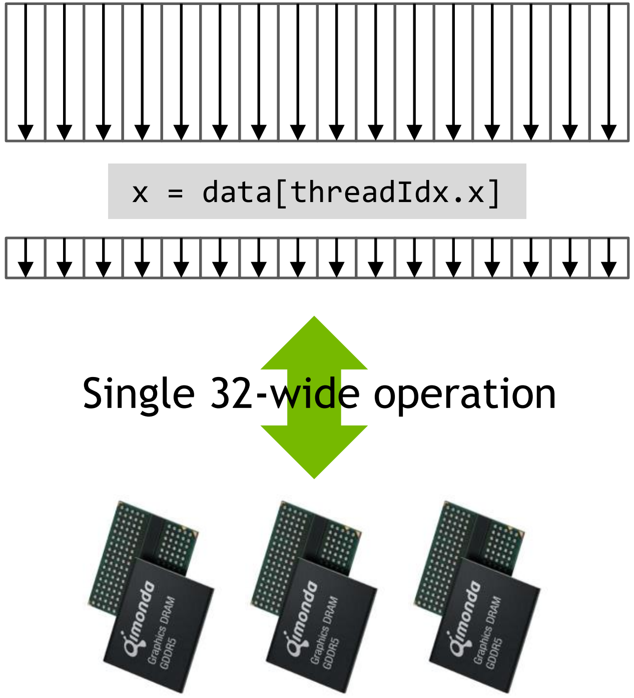
Coalesced: Sequential memory accesses are adjacent



Uncoalesced: Sequential memory accesses are unassociated

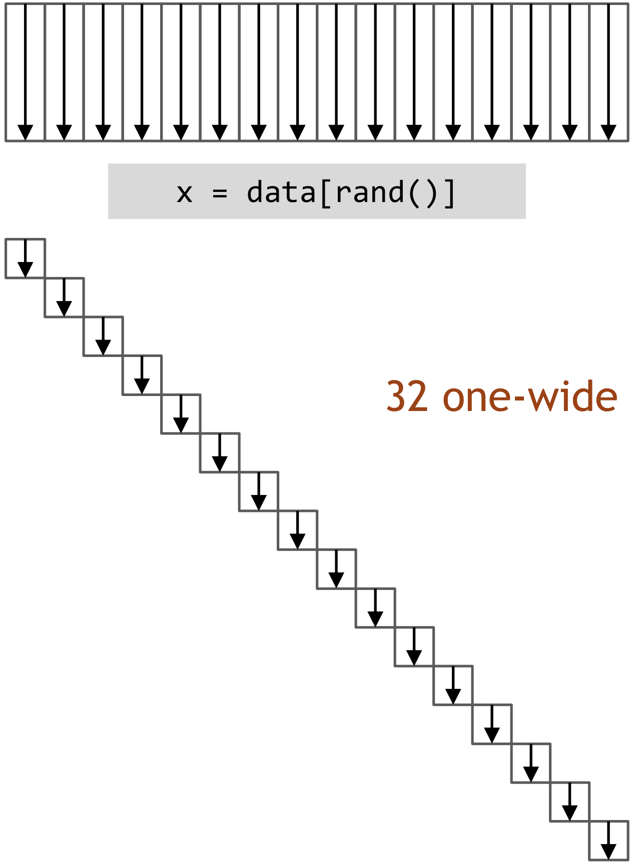


SIMT PENALTIES WHEN NOT COALESCED



`x = data[threadIdx.x]`

Single 32-wide operation

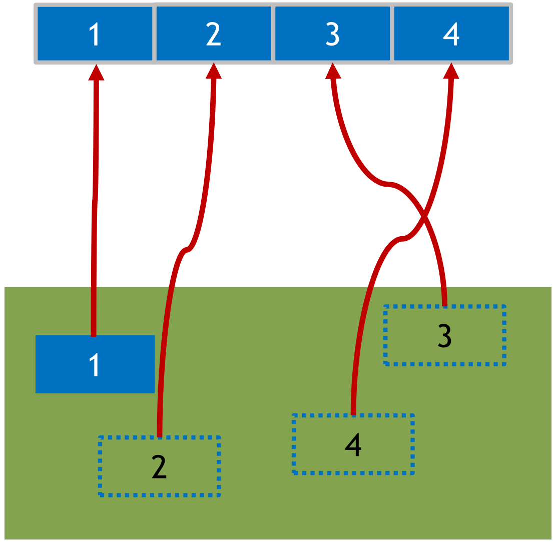


`x = data[rand()]`

32 one-wide operations

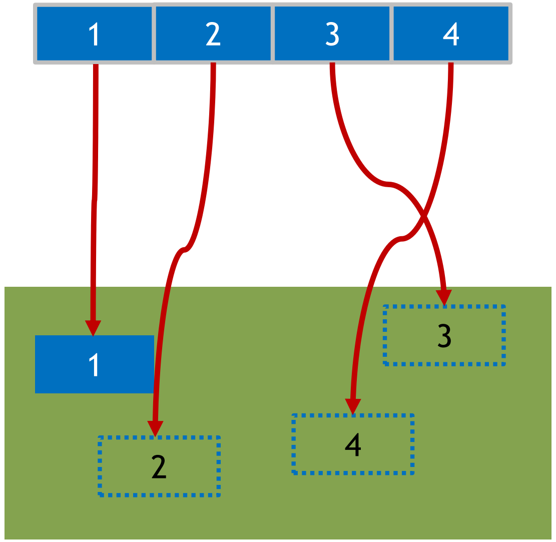
SCATTER & GATHER

Gathering



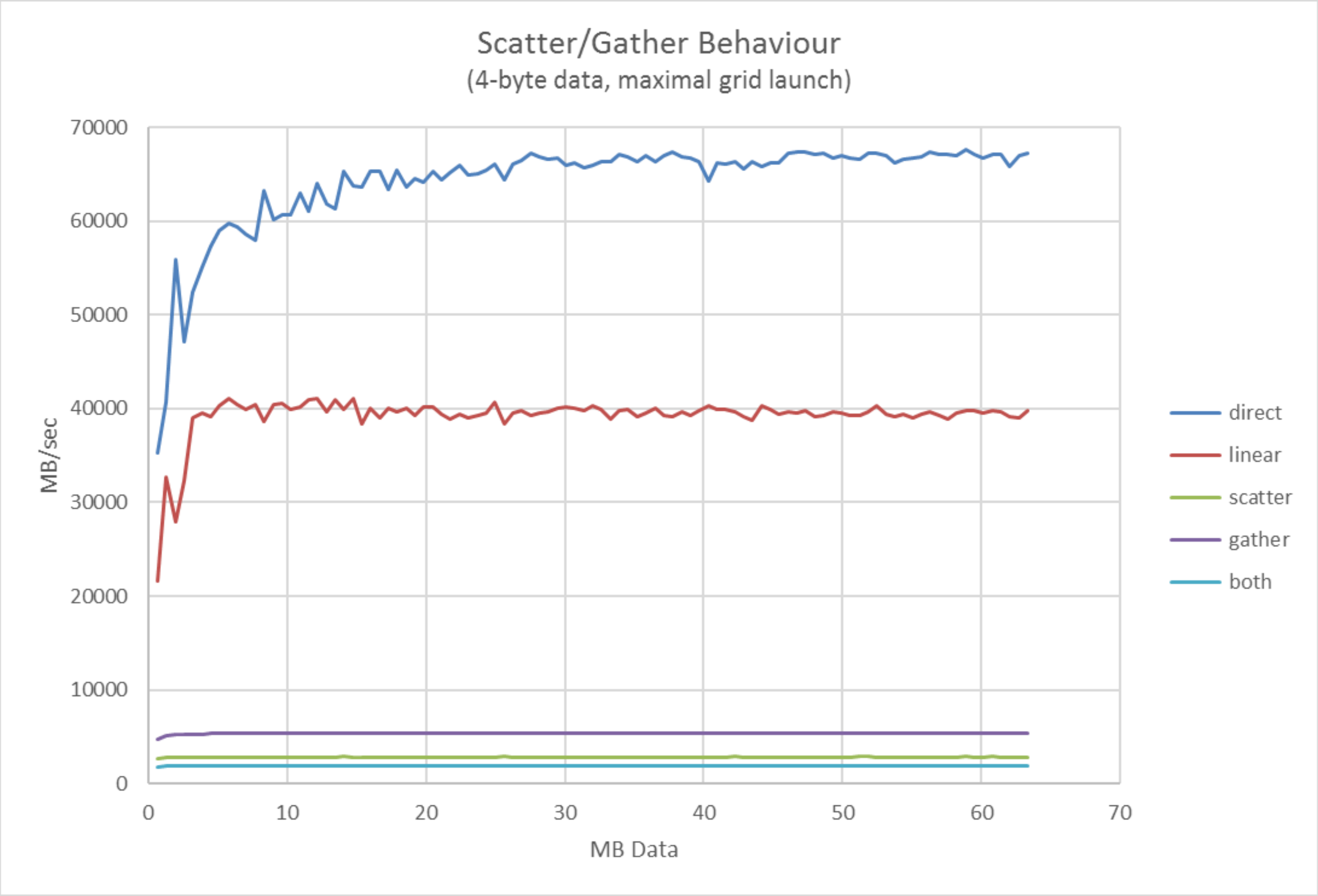
Reading randomly
Writing sequentially

Scattering

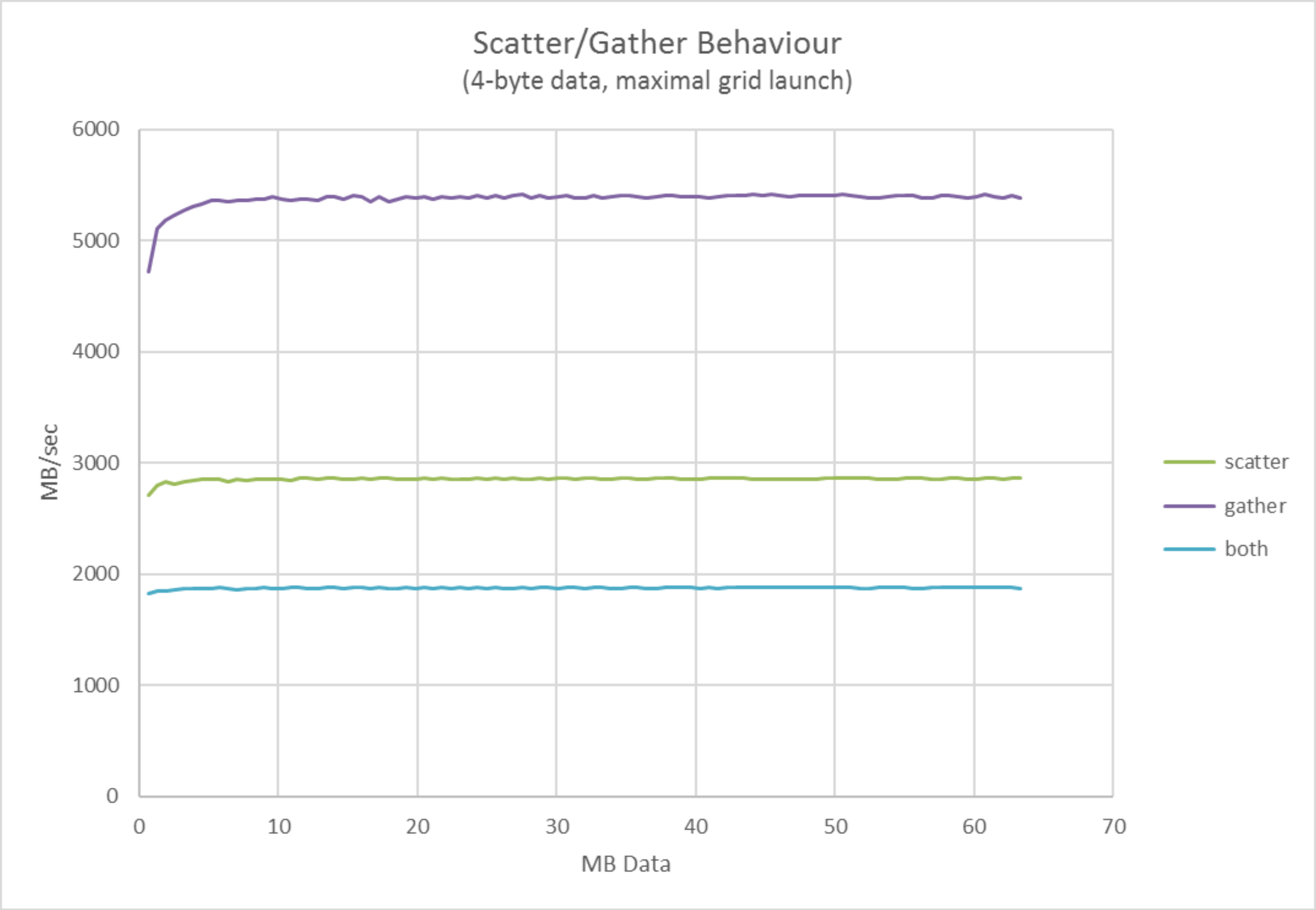


Reading sequentially
Writing randomly

AVOID SCATTER/GATHER IF YOU CAN

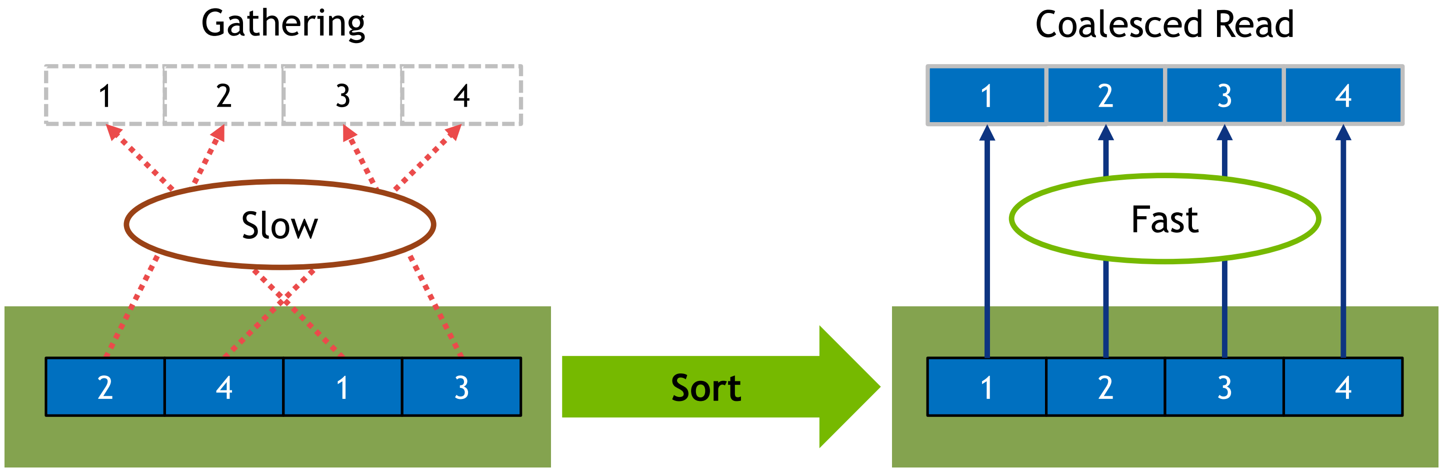


AVOID SCATTER/GATHER IF YOU CAN



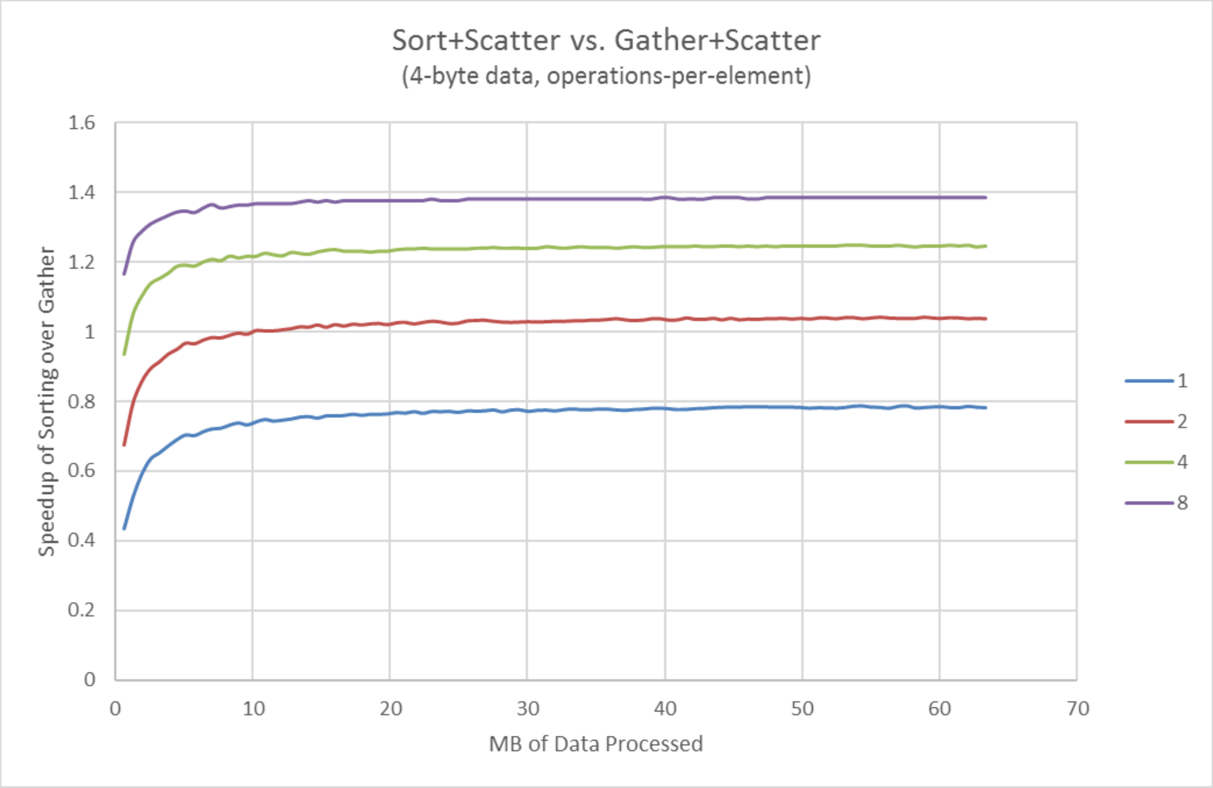
SORTING MIGHT BE AN OPTION

If reading non-sequential data is expensive, is it worth sorting it to make it sequential?

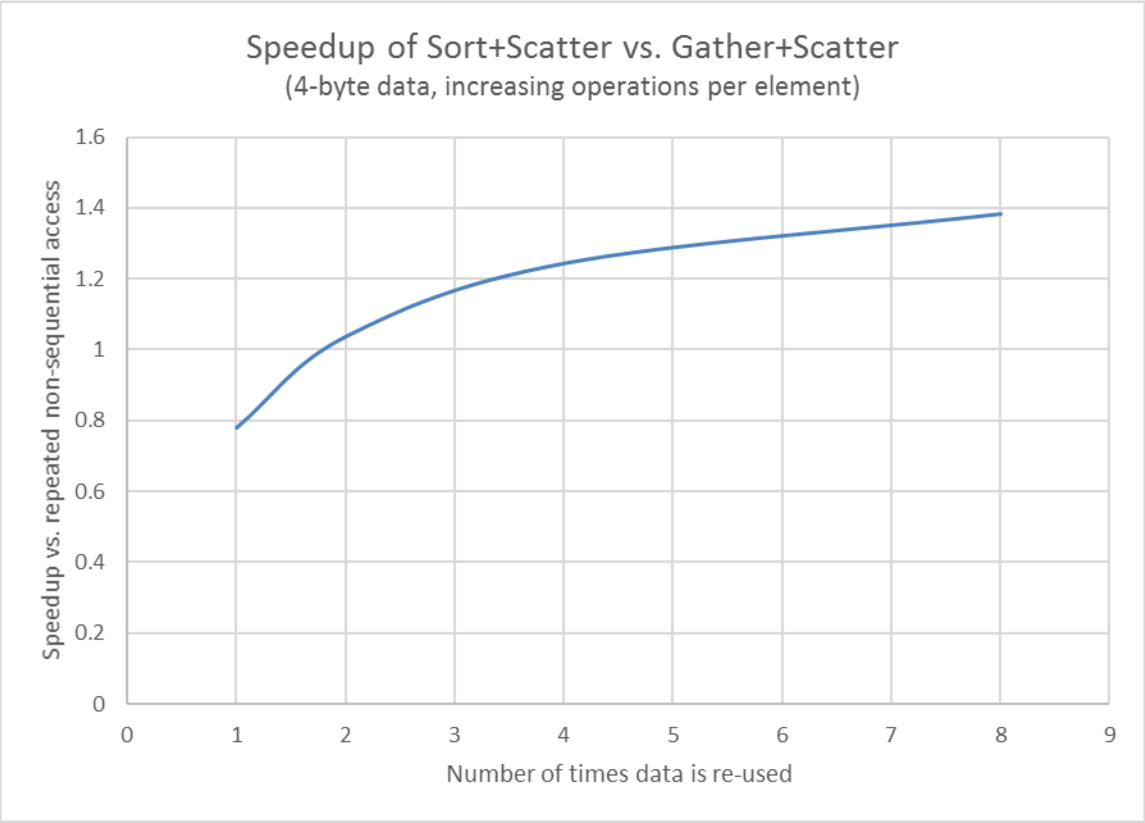


SORTING MIGHT BE AN OPTION

Even if you're only going to read it twice, then yes!



PRE-SORTING TURNS OUT TO BE GOOD



DATA LAYOUT: “AOS vs. SOA”

Sometimes you can’t just sort your data

Array-of-Structures

```
#define NPTS 1024 * 1024

struct Coefficients_AOS {
     double u[3];
     double x[3][3];
     double p;
     double rho;
     double eta;
};

Coefficients_AOS gridData[NPTS];
```

Single-thread code prefers arrays of structures, for cache efficiency

Structure-of-Arrays

```
#define NPTS 1024 *1024

struct Coefficients_SOA {
     double u[3][NPTS];
     double x[3][3][NPTS];
     double p[NPTS];
     double rho[NPTS];
     double eta[NPTS];
};

Coefficients_SOA gridData;
```

SIMT code prefers structures of arrays, for execution & memory efficiency

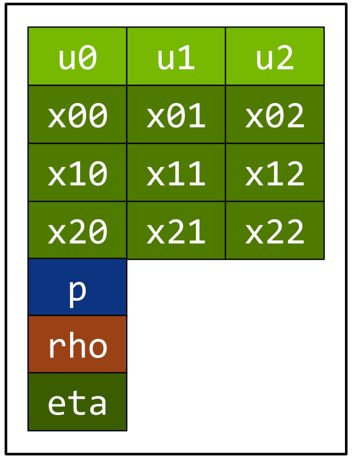
DATA LAYOUT: “AOS vs. SOA”

```
#define NPTS 1024 * 1024

struct Coefficients_AOS {
        double u[3];
        double x[3][3];
        double p;
        double rho;
        double eta;
};

Coefficients_AOS gridData[NPTS];
```

Structure Definition



Conceptual Layout

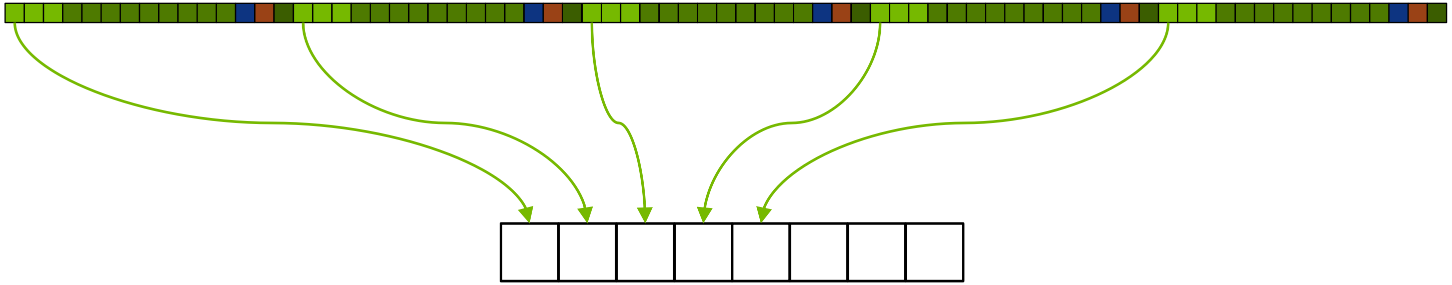
SOA: STRIDED ARRAY ACCESS

GPU reads data one element at a time, but in parallel by 32 threads in a warp

u0	u1	u2
x00	x01	x02
x10	x11	x12
x20	x21	x22
p		
rho		
eta		

Conceptual Layout

Array-of-Structures Memory Layout



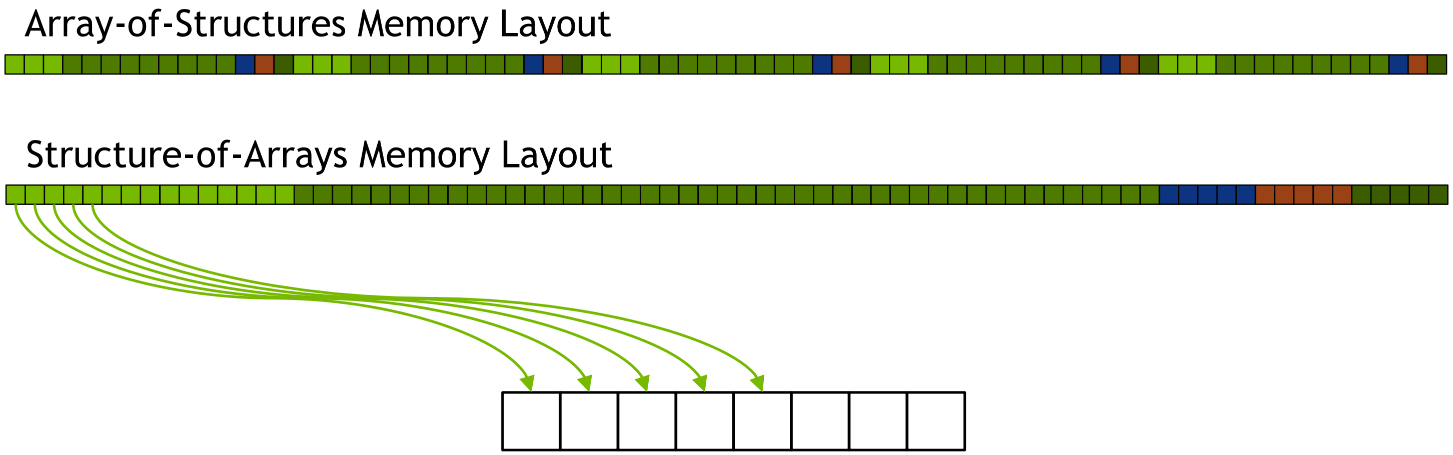
```
double u0 = gridData[threadIdx.x].u[0];
```

AOS: COALESCED BUT COMPLEX

GPU reads data one element at a time, but in parallel by 32 threads in a warp

u0	u1	u2
x00	x01	x02
x10	x11	x12
x20	x21	x22
p		
rho		
eta		

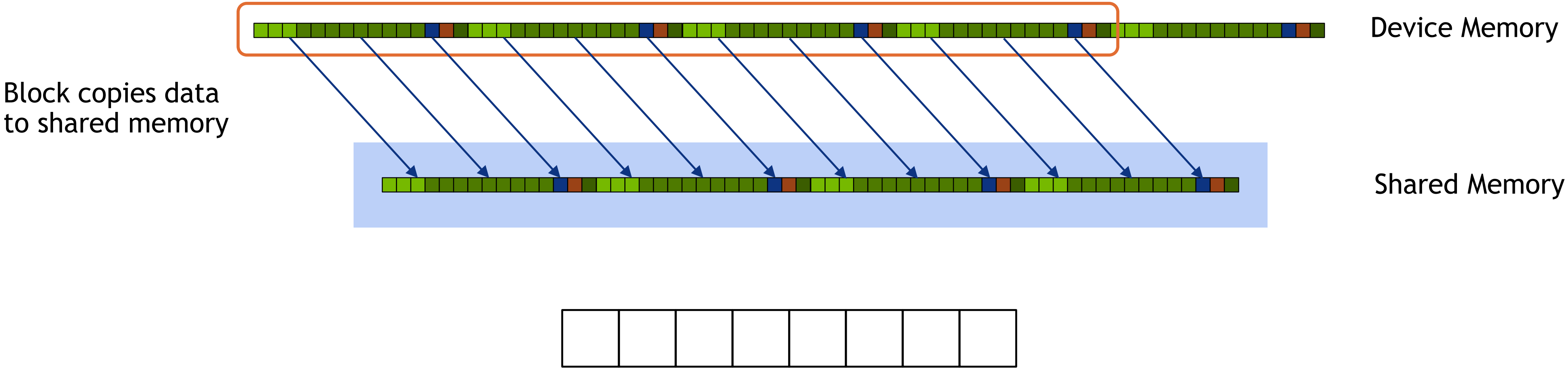
Conceptual Layout


Array-of-Structures Memory Layout
Structure-of-Arrays Memory Layout

```
double u0 = gridData.u[0][threadIdx.x];
```

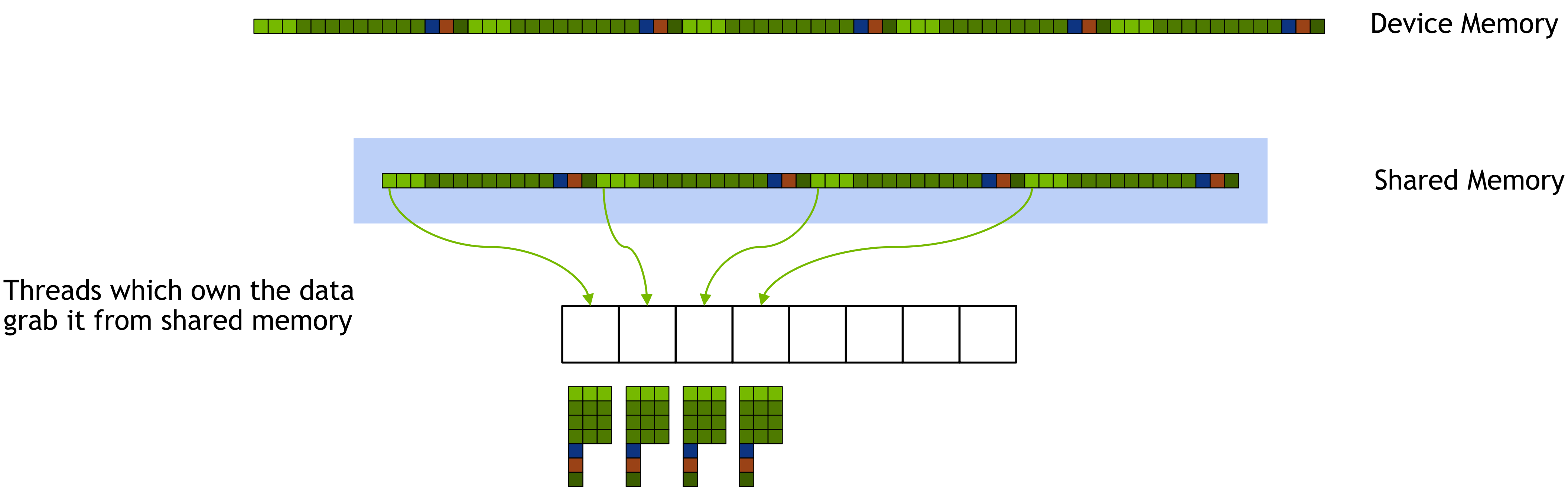
BLOCK-WIDE LOAD VIA SHARED MEMORY

Read data linearly as bytes. Use shared memory to convert to struct

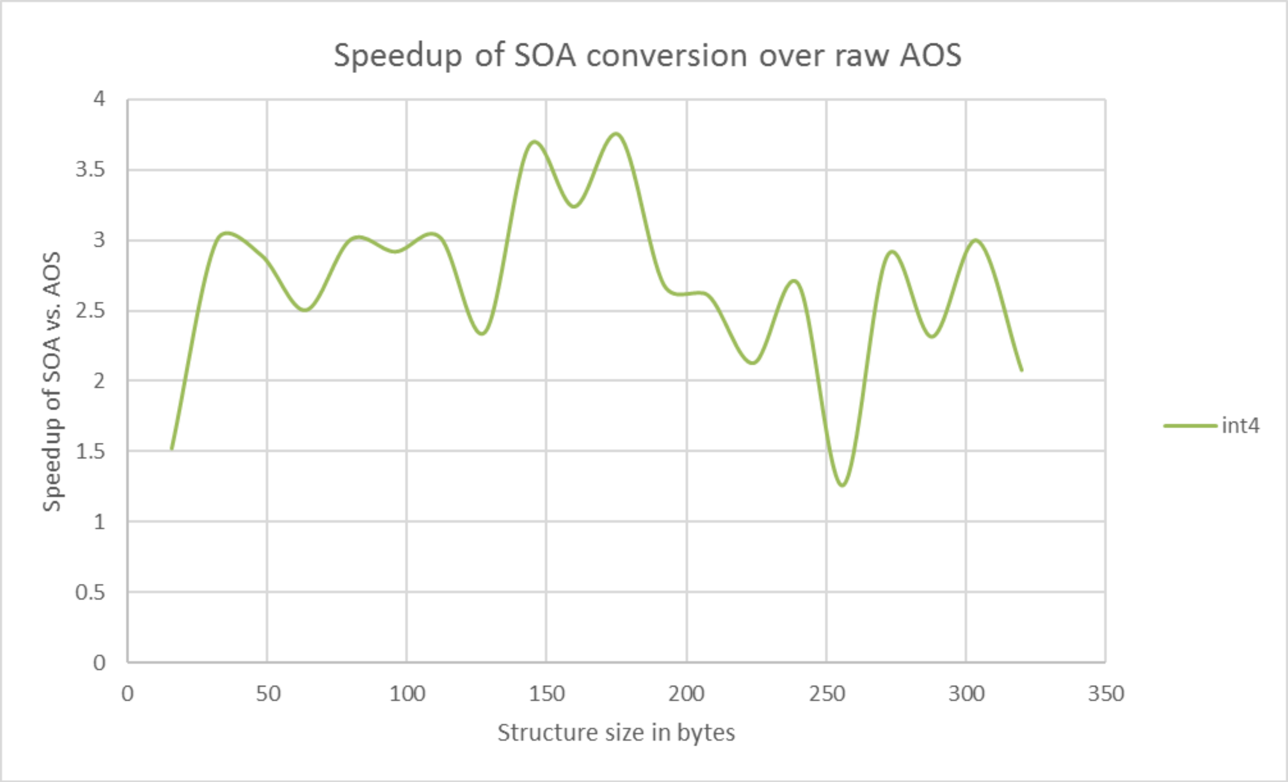


BLOCK-WIDE LOAD VIA SHARED MEMORY

Read data linearly as bytes. Use shared memory to convert to struct

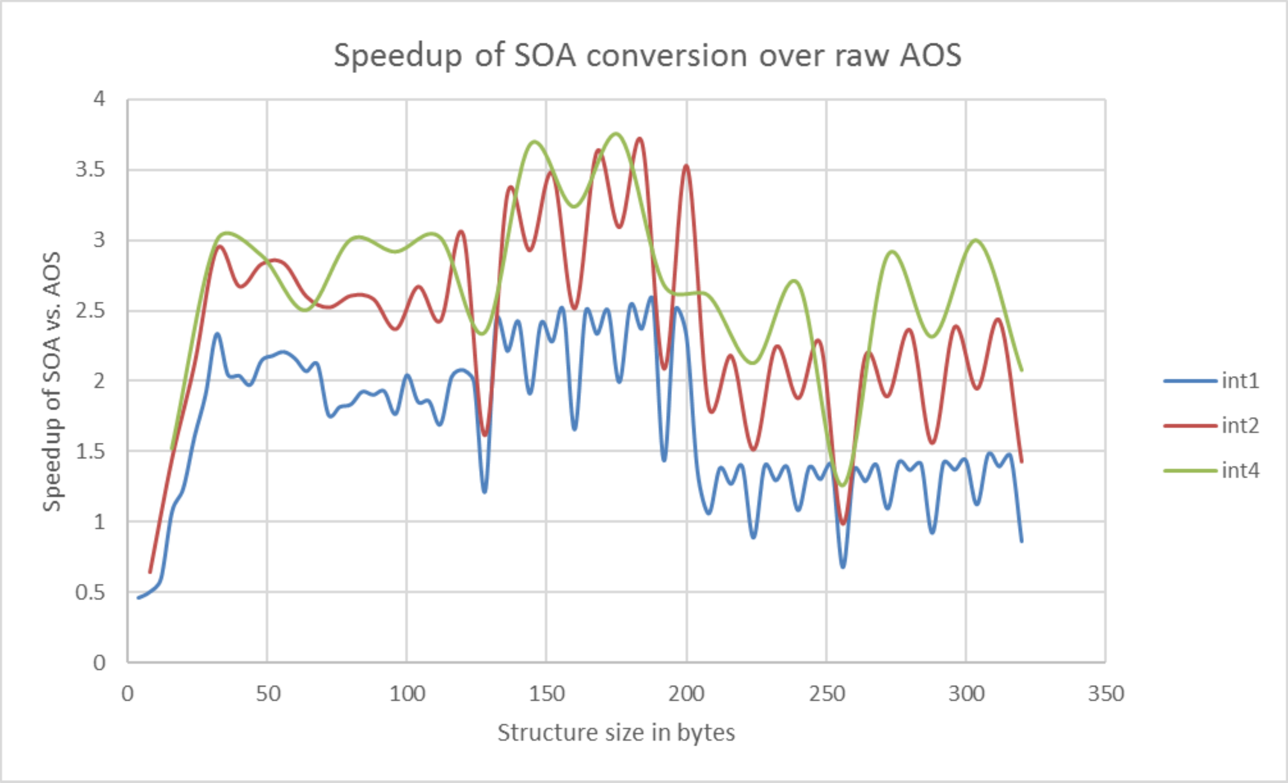


CLEVER AOS/SOA TRICKS



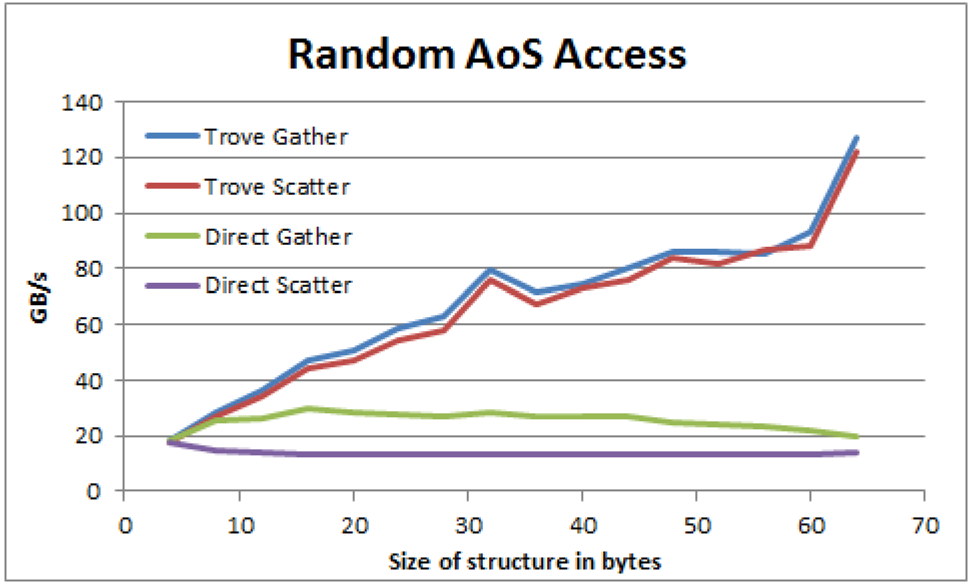
CLEVER AOS/SOA TRICKS

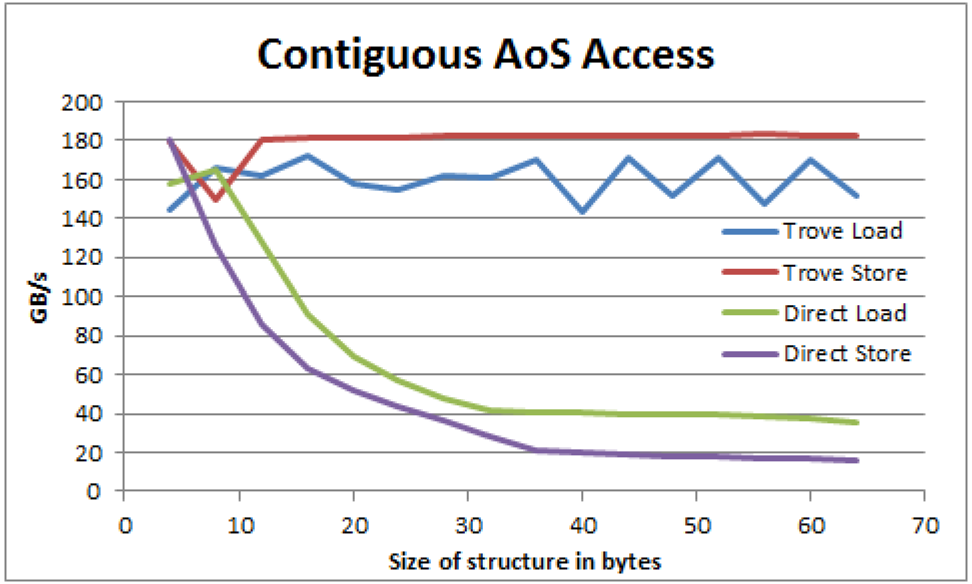
Helps for any data size



HANDY LIBRARY TO HELP YOU

Trove – A utility library for fast AOS/SOA access and transposition
https://github.com/bryancatanzaro/trove





(AB)USING THE CACHE

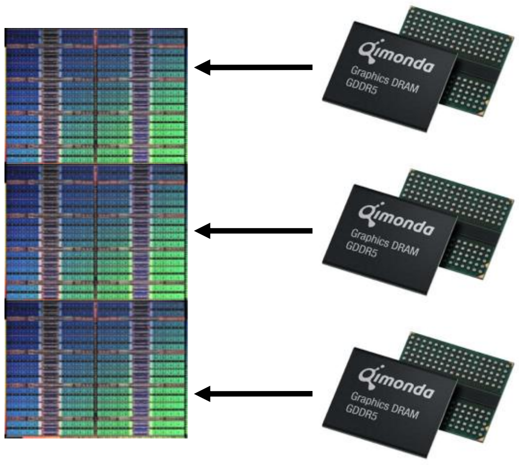
MAKING THE MOST OF L2-CACHE

L2 cache is **fast** but small:

Architecture	L2 Cache Size	Total Threads	Cache Bytes per Thread
Kepler	1536 KB	30,720	51
Maxwell	3072 KB	49,152	64
Pascal	4096 KB	114,688	36

L2 Cache

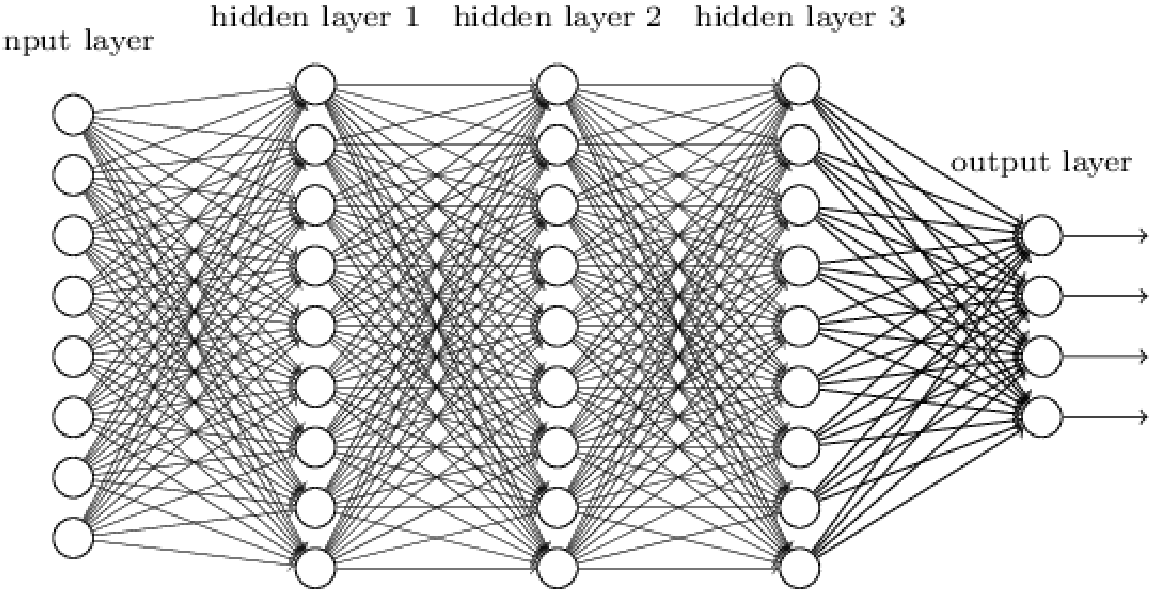
GDRAM



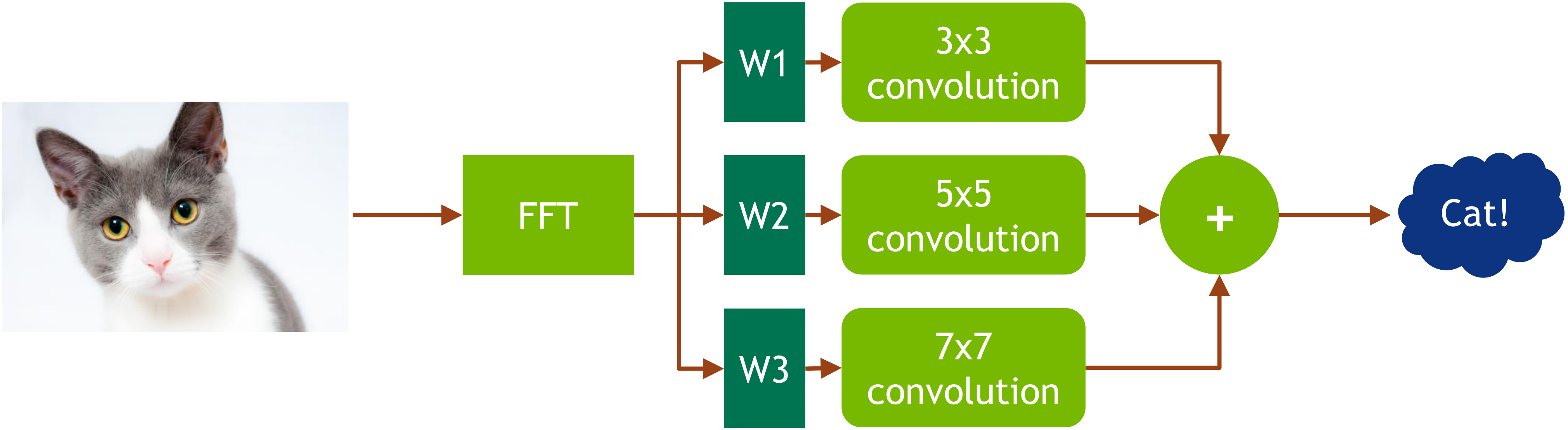
2,000 GB/sec

300 GB/sec

TRAINING DEEP NEURAL NETWORKS



LOTS OF PASSES OVER DATA



FFT

W1

3x3 convolution

W2

5x5 convolution

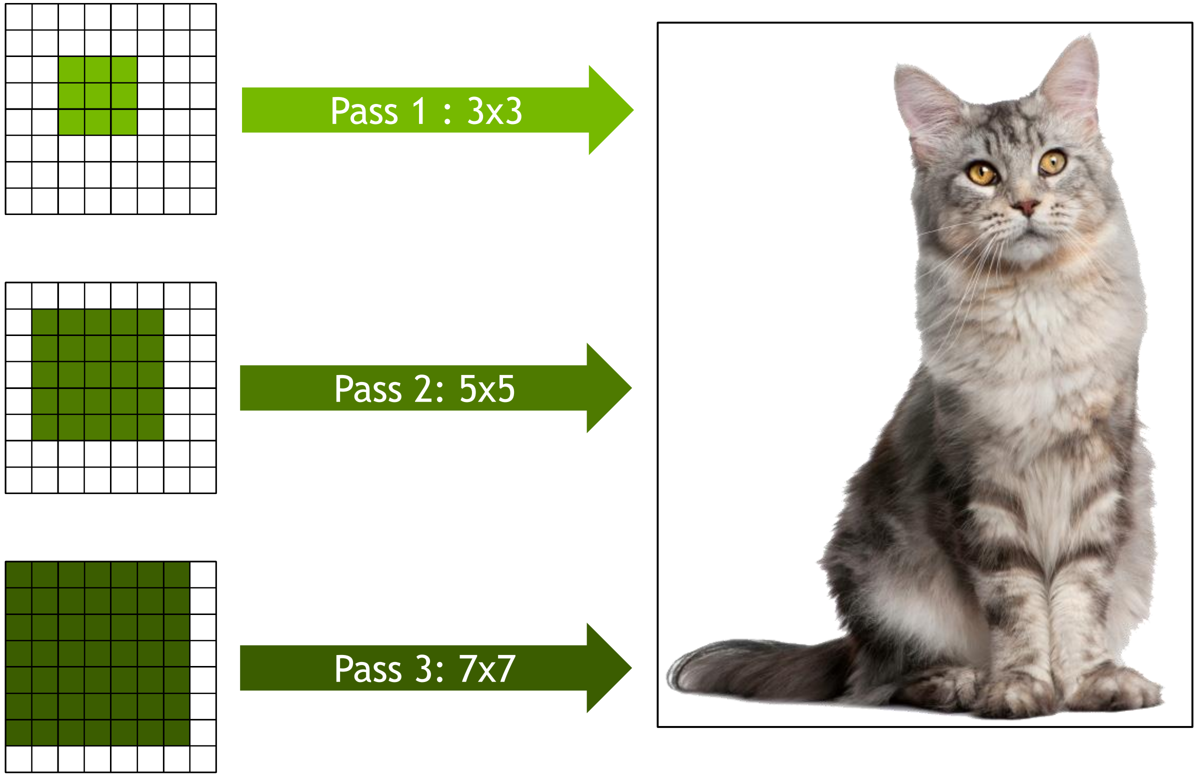
W3

7x7 convolution

+

Cat!

MULTI-RESOLUTION CONVOLUTIONS

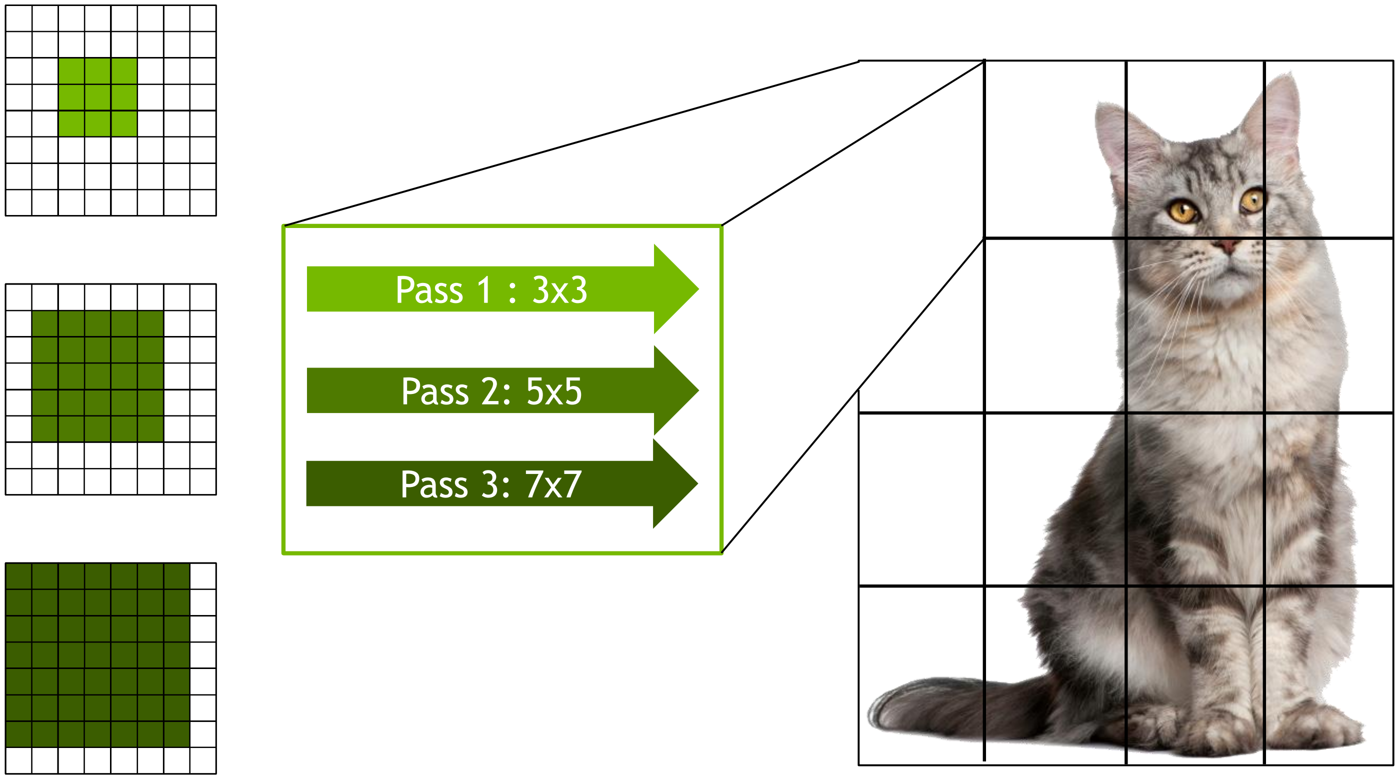


Pass 1 : 3x3

Pass 2: 5x5

Pass 3: 7x7

TILED, MULTI-RESOLUTION CONVOLUTION



Pass 1 : 3x3

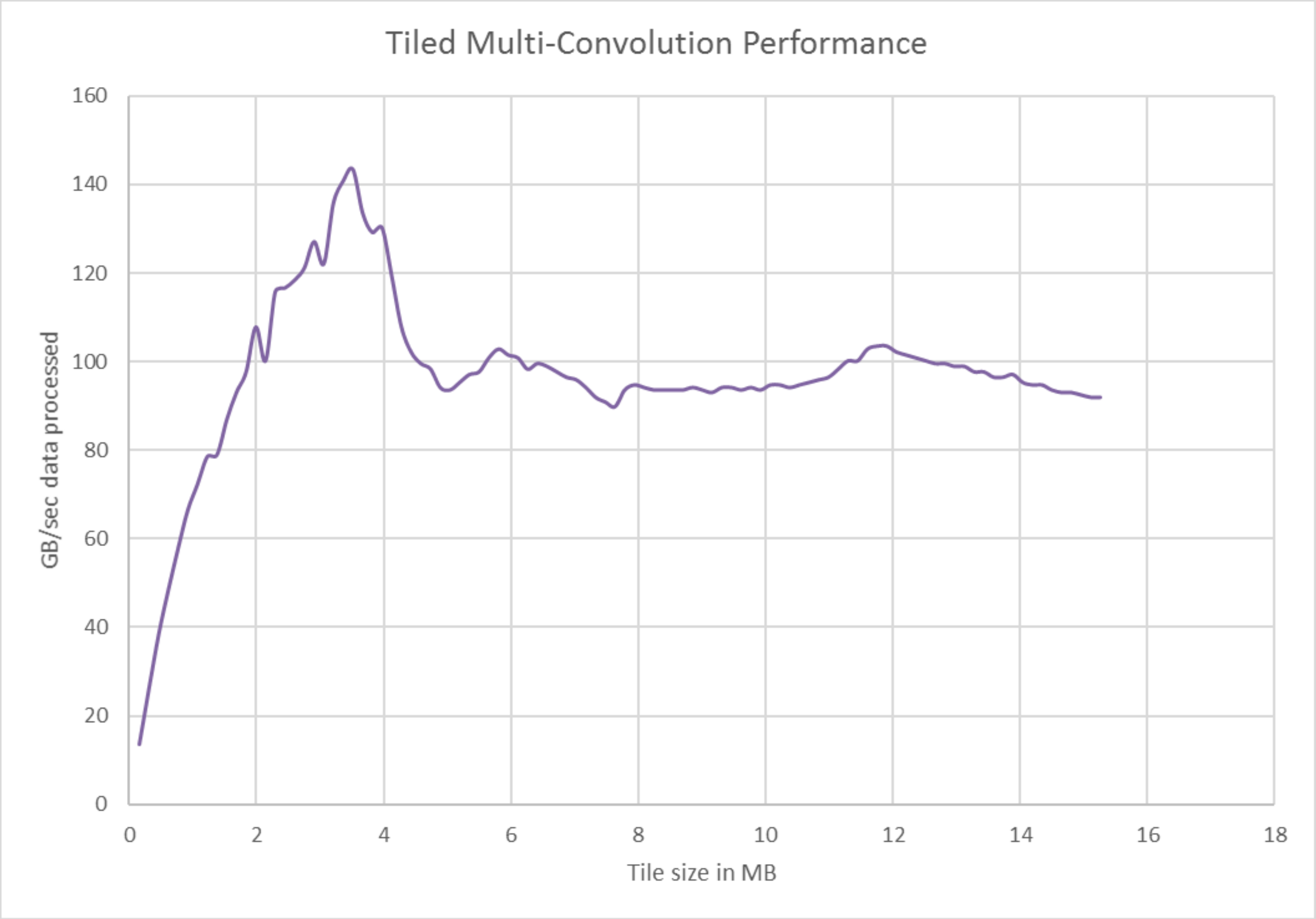
Pass 2: 5x5

Pass 3: 7x7

Do 3 passes per-tile

Each tile sized to fit in L2 cache

LAUNCHING FEWER THAN MAXIMUM THREADS



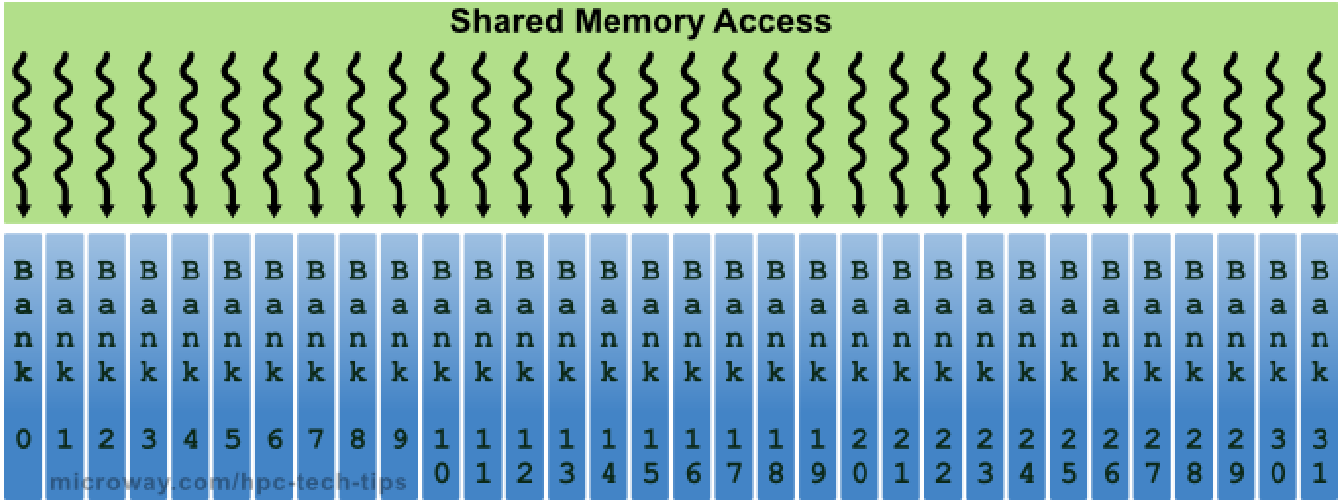
SHARED MEMORY: DEFINITELY WORTH IT

USING SHARED MEMORY WISELY

Shared memory arranged into "banks" for concurrent SIMT access

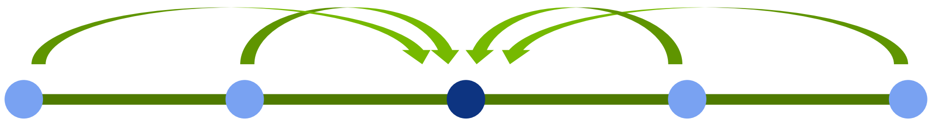
- 32 threads can read simultaneously so long as into separate banks

Shared memory has 4-byte and 8-byte "bank" sizes



STENCIL ALGORITHM

Many algorithms have high data re-use: potentially good for shared memory

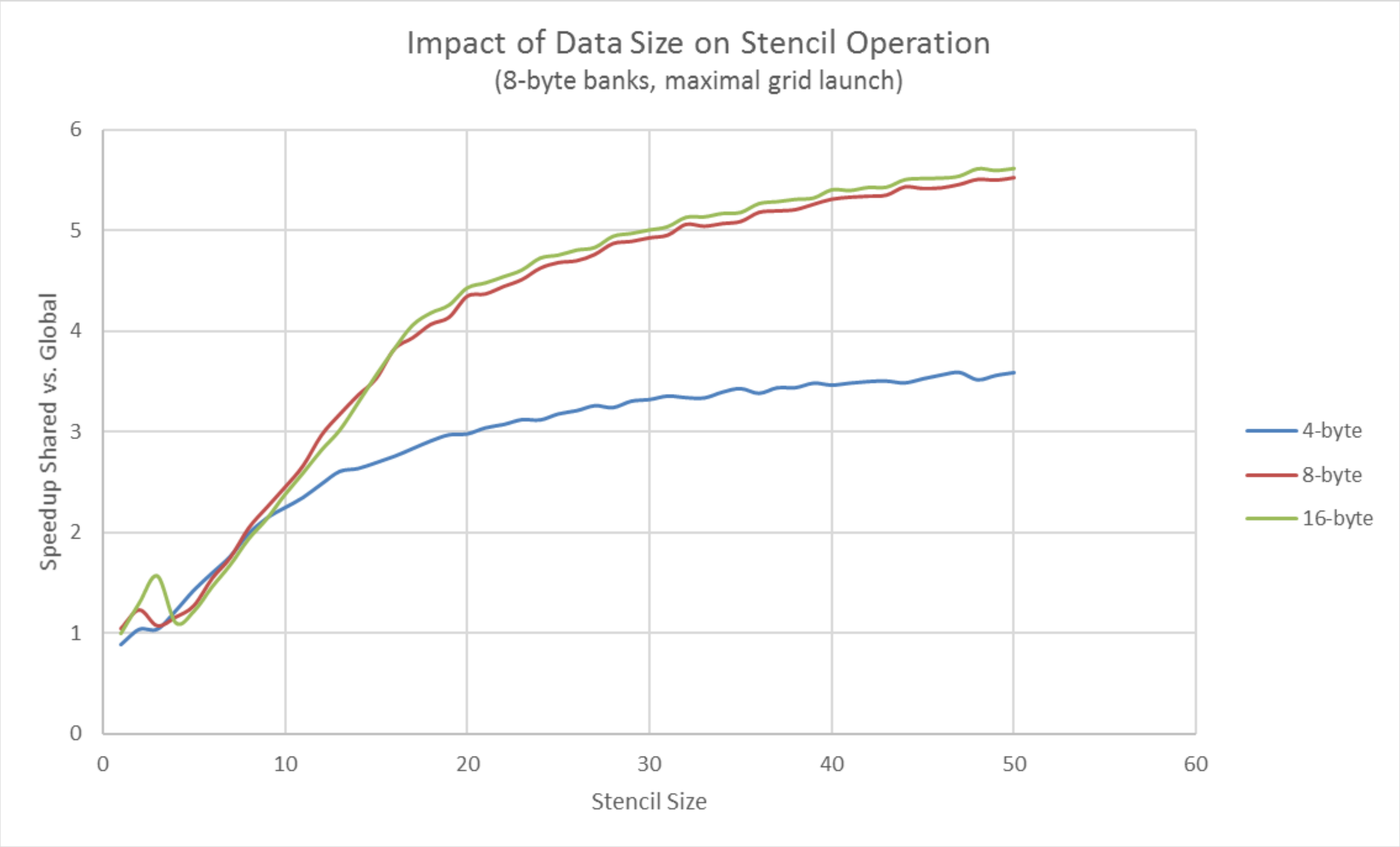


"Stencil" algorithms accumulate data from neighbours onto a central point

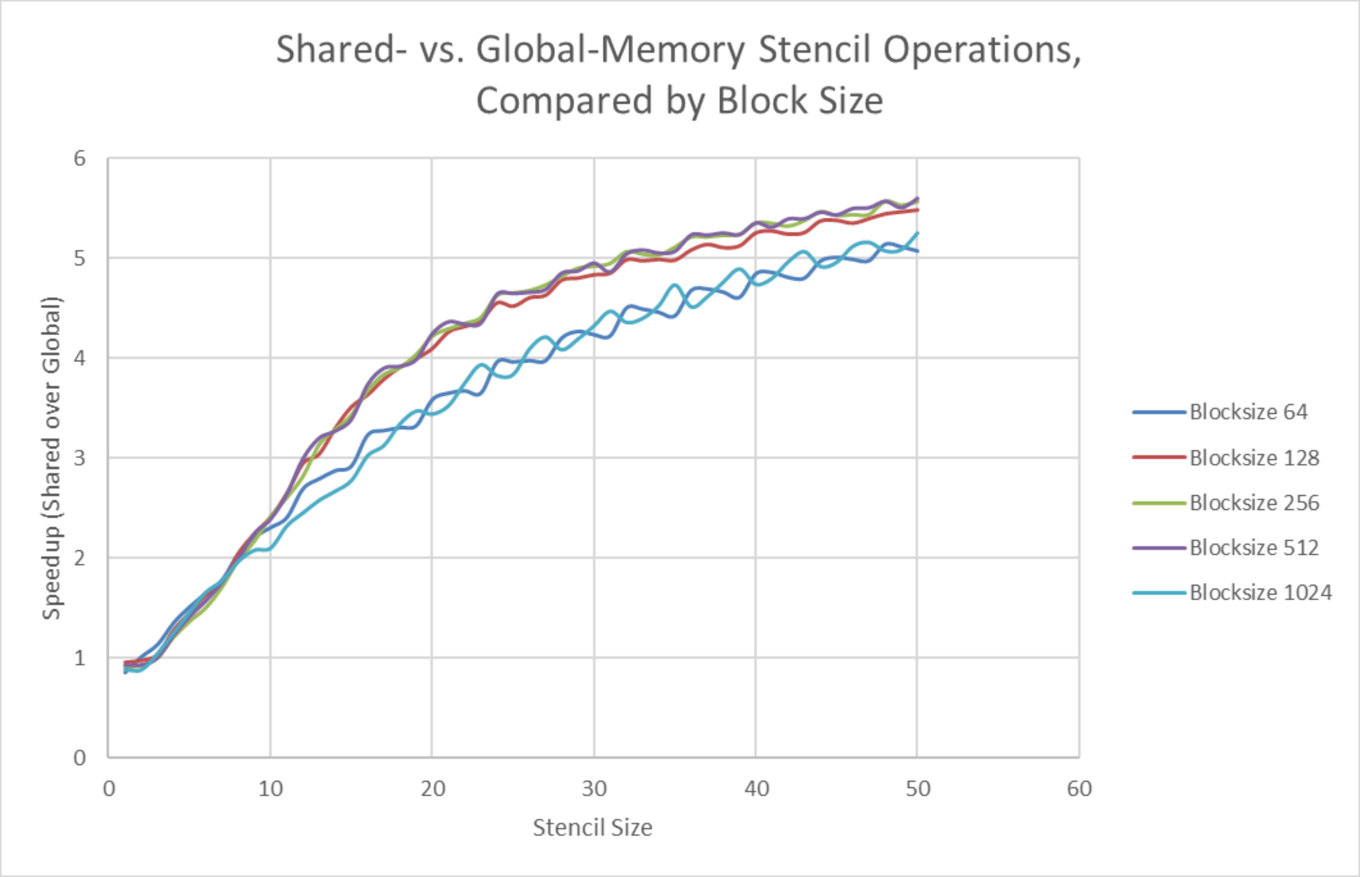
- Stencil has width "W" (in the above case, W=5)

Adjacent threads will share (W-1) items of data – good potential for data re-use

STENCILS IN SHARED MEMORY



SIZE MATTERS



PERSISTENT KERNELS

Revisiting the tiled convolutions

Avoid multiple kernel launches by caching in shared memory instead of L2

```
void tiledConvolution() {
      convolution<3><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
      convolution<5><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
      convolution<7><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
}
```

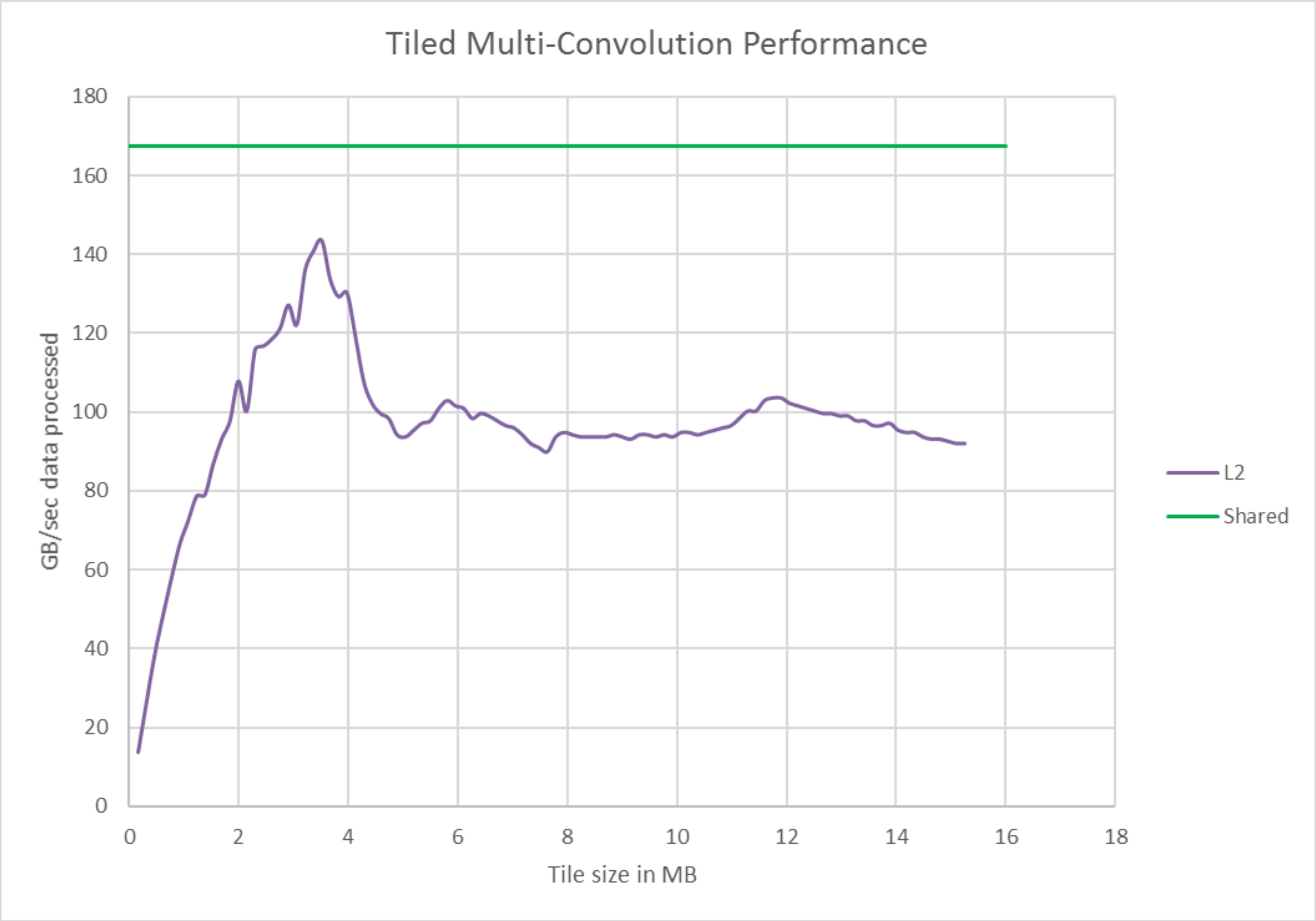
Separate kernel launches with L2 re-use

```
__global__ void convolutionShared(int *data, int count, int sharedelems) {
      extern __shared__ int shdata[];
      shdata[threadIdx.x] = data[threadIdx.x + blockDim.x*blockIdx.x];
      __syncthreads();

      convolve<3>(threadIdx.x, shdata, sharedelems);
      __syncthreads();
      convolve<5>(threadIdx.x, shdata, sharedelems);
      __syncthreads();
      convolve<7>(threadIdx.x, shdata, sharedelems);
}
```

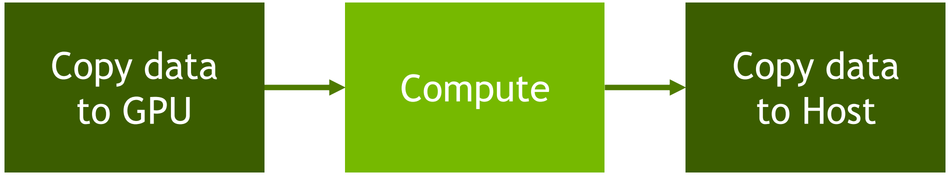
Single kernel launch with persistent kernel

PERSISTENT KERNELS

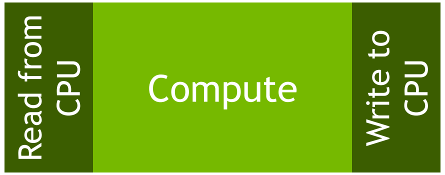


OPERATING DIRECTLY FROM CPU MEMORY

Can save memory copies. It’s obvious when you think about it ...

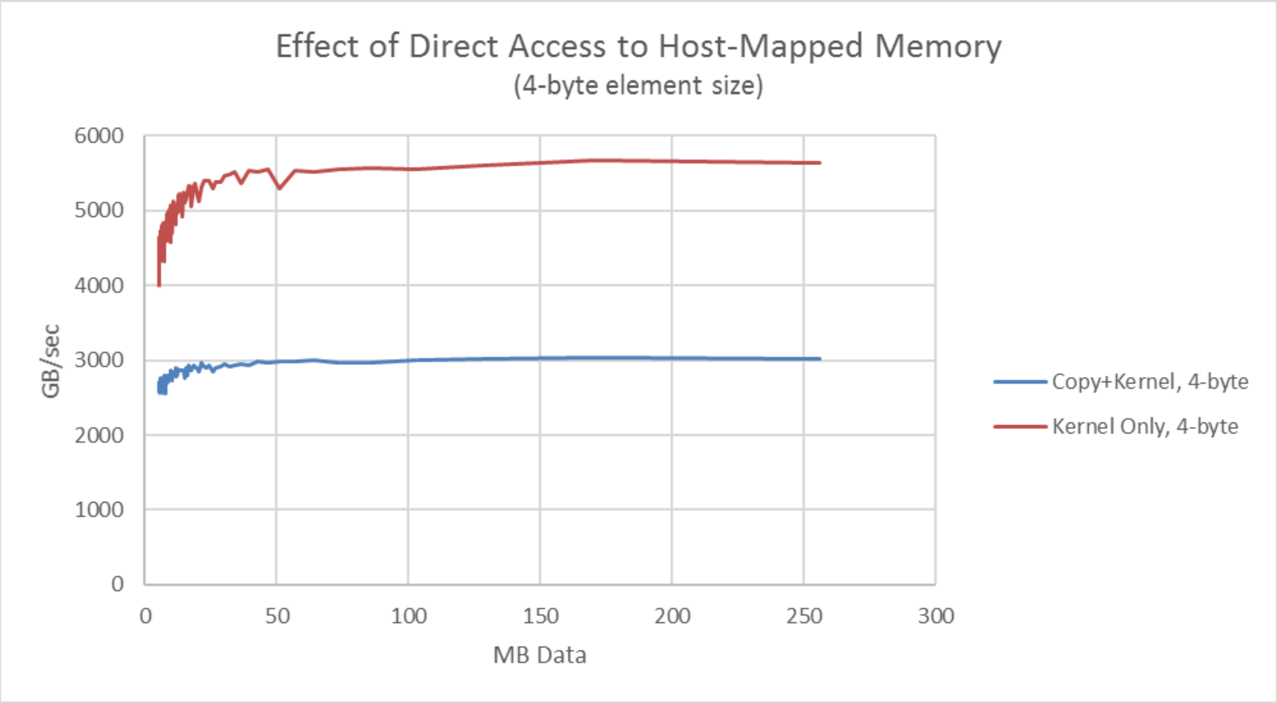


Compute only begins when 1st copy has finished. Task only ends when 2nd copy has finished.



Compute begins after first fetch. Uses lots of threads to cover host-memory access latency. Takes advantage of bi-directional PCI.

OPERATING DIRECTLY FROM CPU MEMORY



OCCUPANCY AND REGISTER LIMITATIONS

Register file is bigger than shared memory and L1 cache!

Occupancy can kill you if you use too many registers

Often worth forcing fewer registers to allow more blocks per SM

But watch out for math functions!

```
__launch_bounds__(maxThreadsPerBlock,
                  minBlocksPerMultiprocessor)

__global__ void compute() {
    y = acos(pow(log(fdivide(tan(cosh(erfc(x))), 2)), 3);
}
```

Function	float	double
log	7	18
cos	16	28
acos	6	18
cosh	7	10
tan	15	28
erfc	14	22
exp	7	10
log10	6	18
normcdf	16	26
cbrt	8	20
sqrt	6	12
rsqrt	5	12
y0	20	30
y1	22	30
fdivide	11	20
pow	11	24
grad. desc.	14	22

THANK YOU!

NVIDIA