An Introduction to CUDA/OpenCL and Manycore Graphics Processors
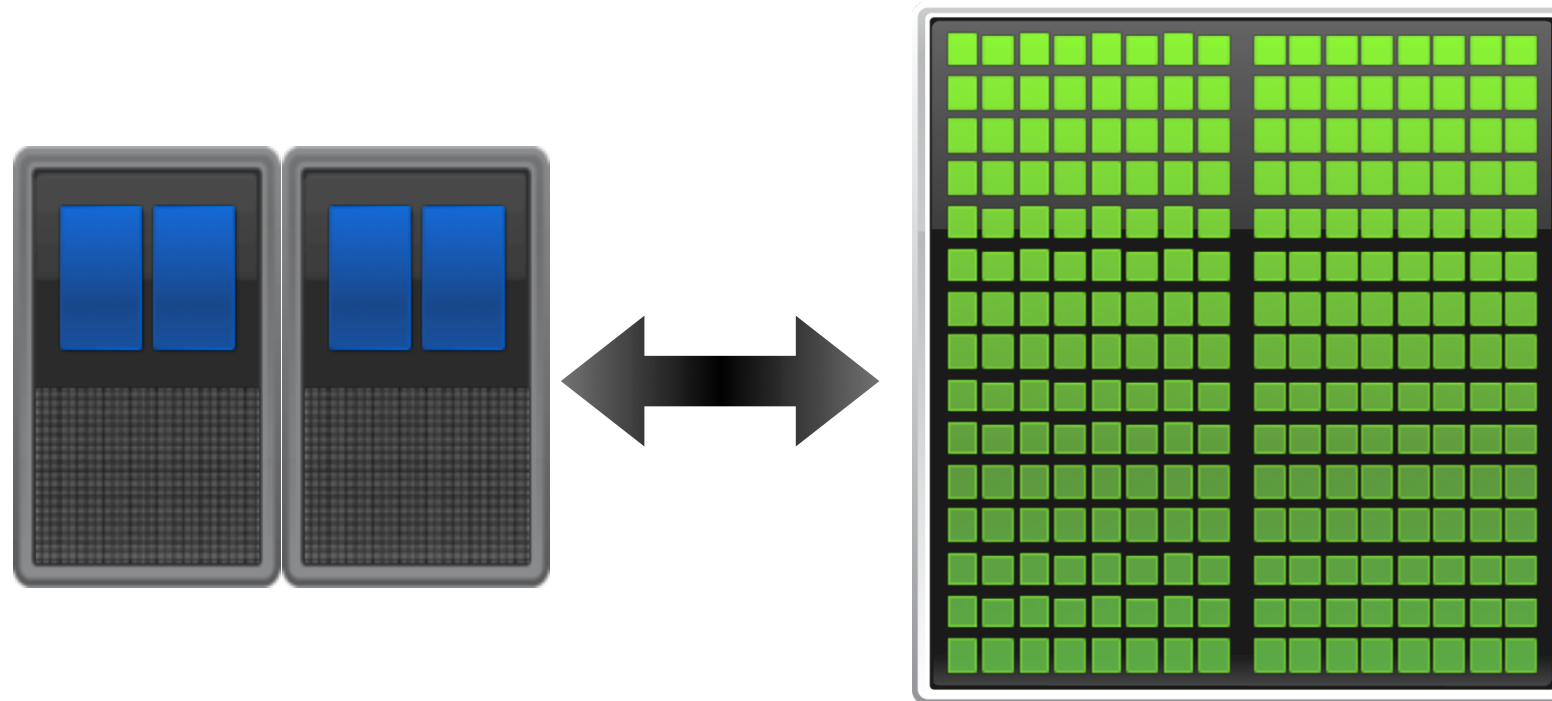
Bryan Catanzaro, NVIDIA Research

Overview

- Terminology: Multicore, Manycore, SIMD
- The CUDA and OpenCL programming models
- Understanding how CUDA maps to NVIDIA GPUs
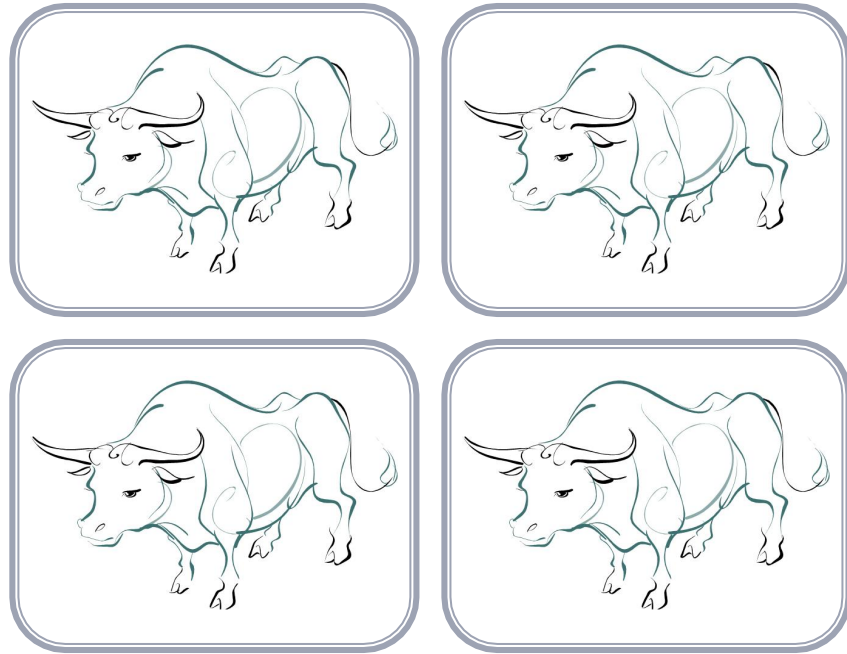- Thrust

Heterogeneous Parallel Computing

Multicore CPU

Fast Serial
Processing

Manycore GPU

Scalable Parallel
Processing

Multicore and Manycore
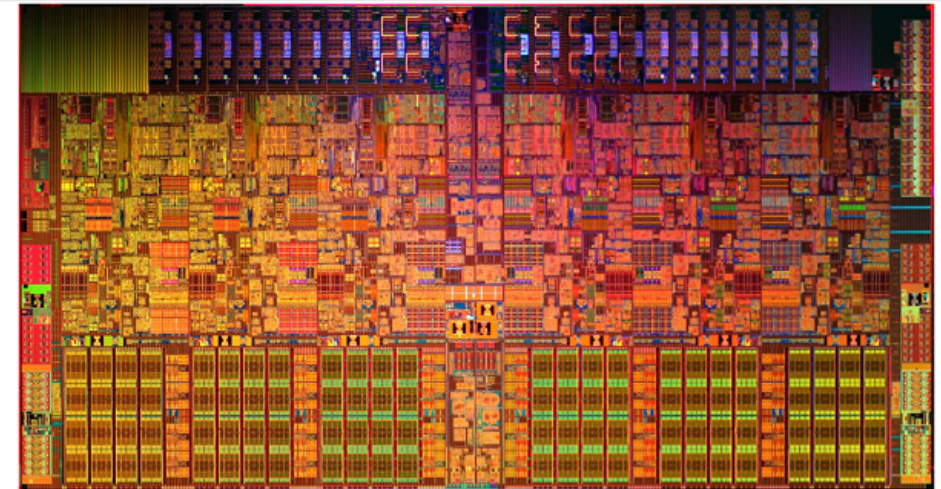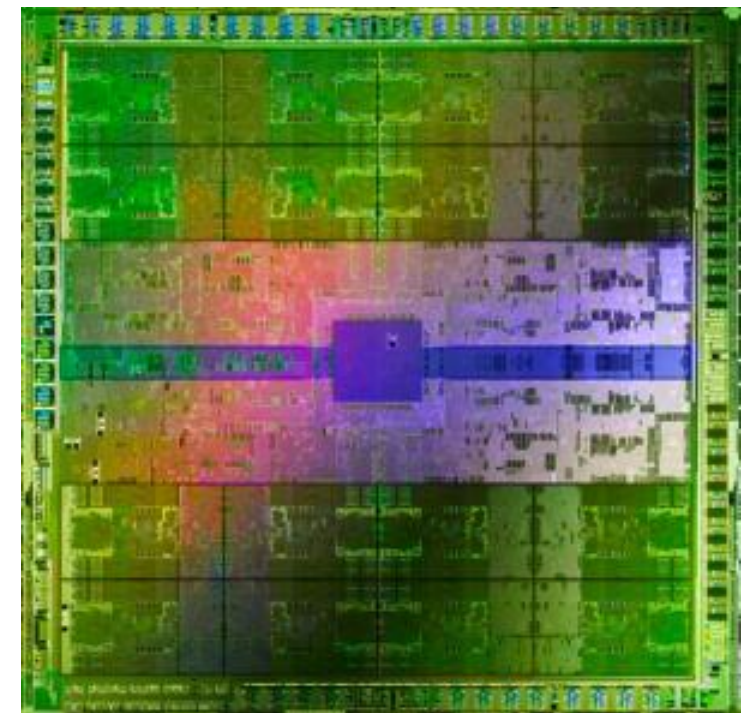
Multicore

Manycore

- Multicore: yoke of oxen

  - Each core optimized for executing a single thread

- Manycore: flock of chickens

  - Cores optimized for aggregate throughput, deemphasizing individual performance

- *(apologies to Seymour Cray)*

Multicore & Manycore, *cont*.

| Specifications	Westmere-EP	Fermi (Tesla C2050)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	14 SMs, 2 issue, 16 way SIMD @1.15 GHz
Resident Strands/ Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads
SP GFLOP/s	166	1030
Memory Bandwidth	32 GB/s	144 GB/s
Register File	6 kB (?)	1.75 MB
Local Store/L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-

Westmere-EP (32nm)

Fermi (40nm)

Why Heterogeneity?

- Different goals produce different designs
  - Manycore assumes work load is highly parallel
  - Multicore must be good at everything, parallel or not

- Multicore: minimize latency experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control

- Manycore: maximize throughput of all threads
  - lots of big ALUs
  - multithreading can hide latency … so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD

SIMD

SISD

a   b

+

c

SIMD
width=2

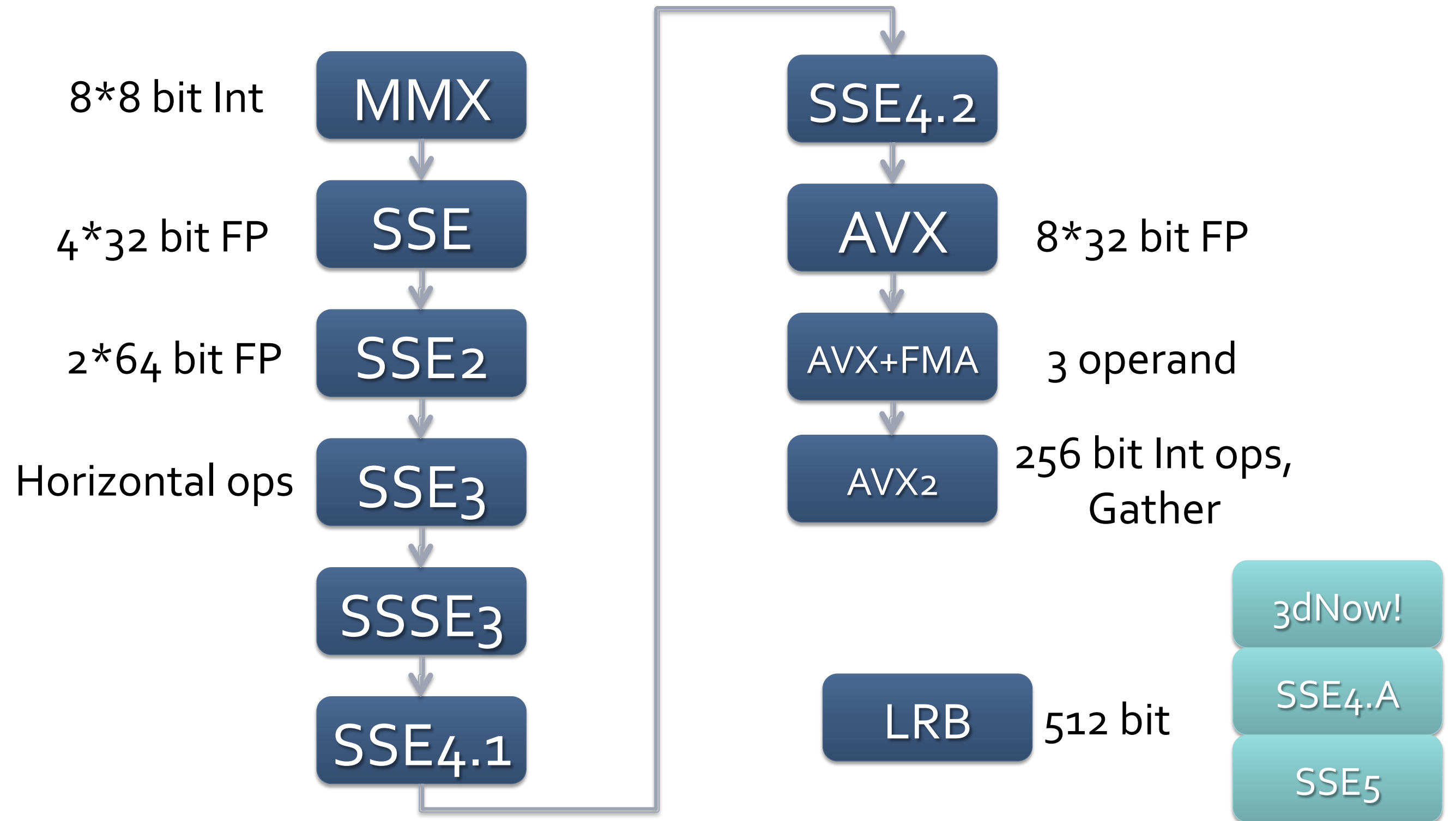a_1 a_2 b_1 b_2

+

c_1 c_2
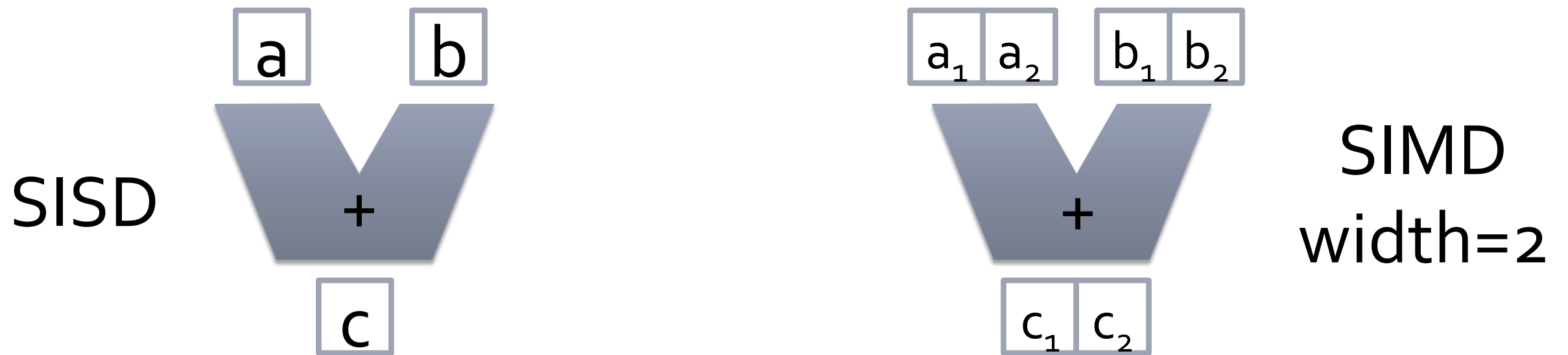
- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

SIMD: Neglected Parallelism

- OpenMP / Pthreads / MPI all neglect SIMD parallelism
- Because it is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?

  - Many languages (like C) are difficult to vectorize

- Most common solution:

  - Either forget about SIMD

    - Pray the autovectorizer likes you

  - Or instantiate intrinsics (assembly language)

  - Requires a new code version for every SIMD extension

A Brief History of x86 SIMD Extensions

8*8 bit Int — **MMX**

4*32 bit FP — **SSE**

2*64 bit FP — **SSE2**

Horizontal ops — **SSE3**

SSSE3

SSE4.1

SSE4.2

AVX — 8*32 bit FP

AVX+FMA — 3 operand

AVX2 — 256 bit Int ops, Gather

LRB — 512 bit

3dNow!

SSE4.A

SSE5

What to do with SIMD?

| Used | Wasted |

4 way SIMD (SSE)   16 way SIMD (LRB)

- Neglecting SIMD is becoming more expensive

  - AVX: 8 way SIMD, Larrabee: 16 way SIMD,
    Nvidia: 32 way SIMD, ATI: 64 way SIMD

- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

The CUDA Programming Model

- CUDA is a programming model designed for:
    - Manycore architectures
    - Wide SIMD parallelism
    - Scalability

- CUDA provides:
    - A thread abstraction to deal with SIMD
    - Synchronization & data sharing between small groups of threads

- CUDA programs are written in C++ with minimal extensions

- OpenCL is inspired by CUDA, but HW & SW vendor neutral
    - Similar programming model, C only for device code

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

Thread t

- Threads are grouped into thread blocks
  - threads in the same block can cooperate

Block b

```
t0 t1 … tN
```

- Threads/blocks have unique IDs
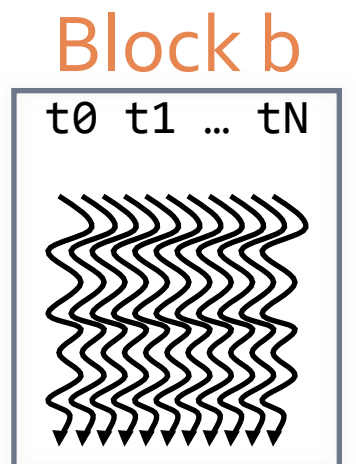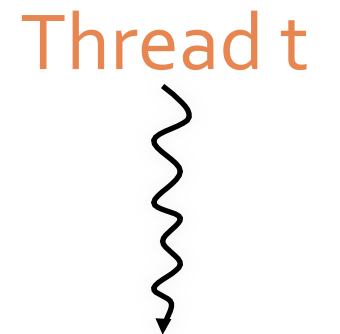
What is a CUDA Thread?

- Independent thread of execution
  - has its own program counter, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

What is a CUDA Thread Block?

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for *any interleaving* of block executions

CUDA Supports:

- Thread parallelism
  - each thread is an independent thread of execution

- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams

Synchronization

- Threads within a block may synchronize with barriers

  ```
  … Step 1 …
  __syncthreads();
  … Step 2 …
  ```

- Blocks coordinate via atomic memory operations

  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels

  ```
  vec_minus<<<nblocks, blksize>>>(a, b, c);
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  vec_dot<<<nblocks, blksize>>>(c, c);
  ```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock

- Independence requirement gives scalability

Scalability

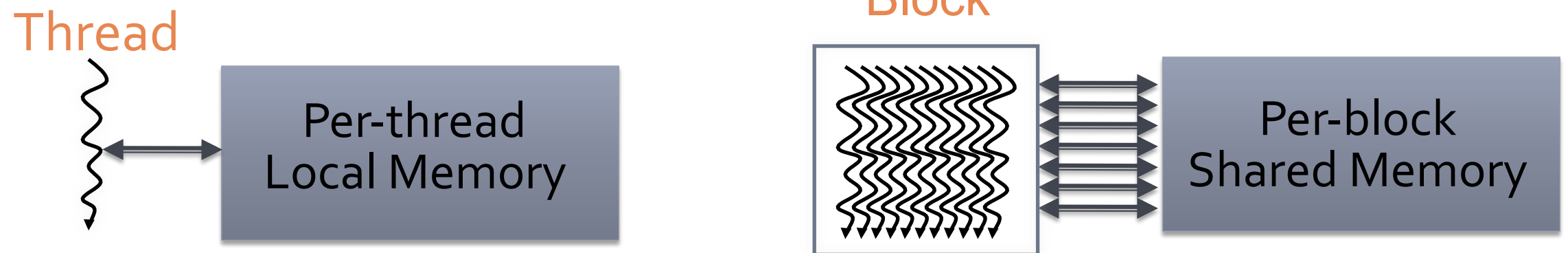- Manycore chips exist in a diverse set of configurations

Number of SMs
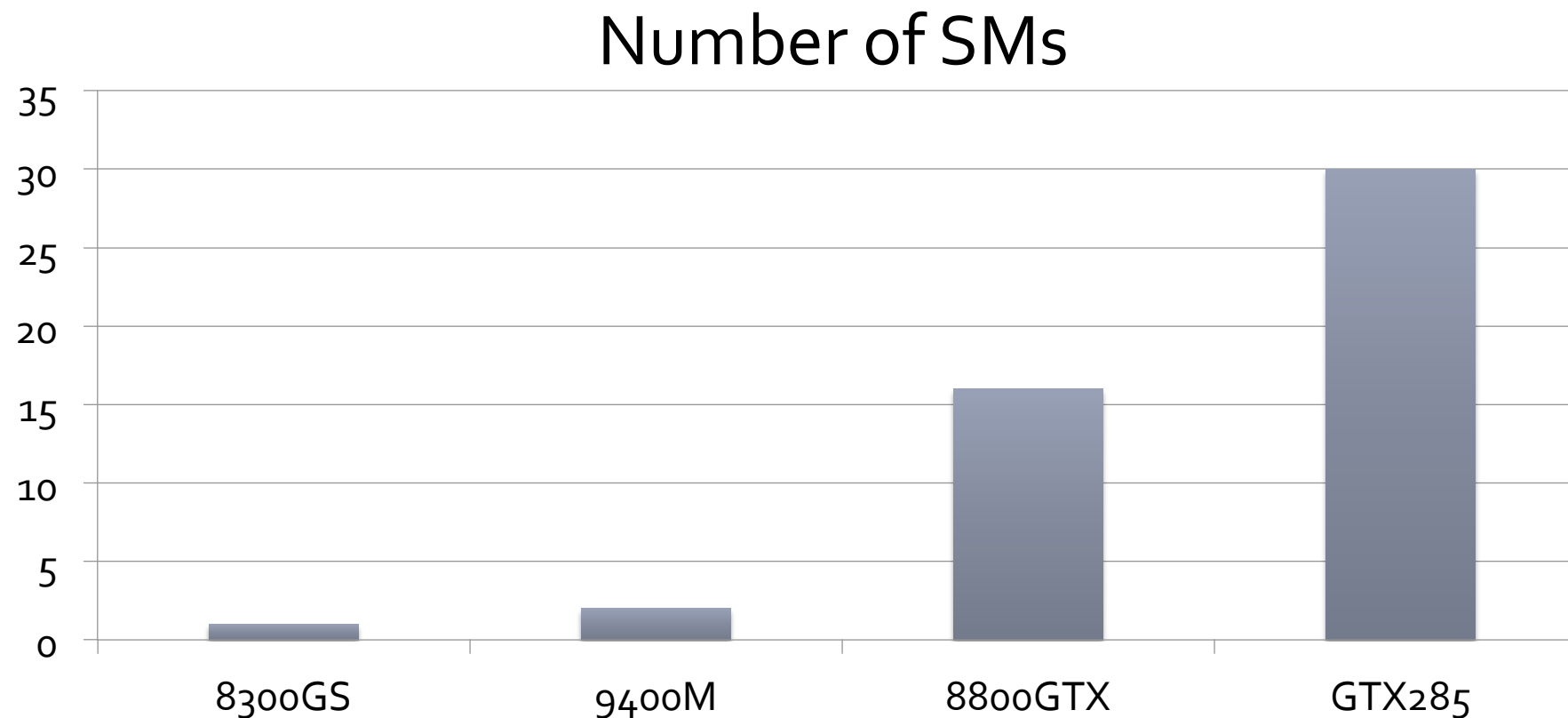
- CUDA allows one binary to target all these chips
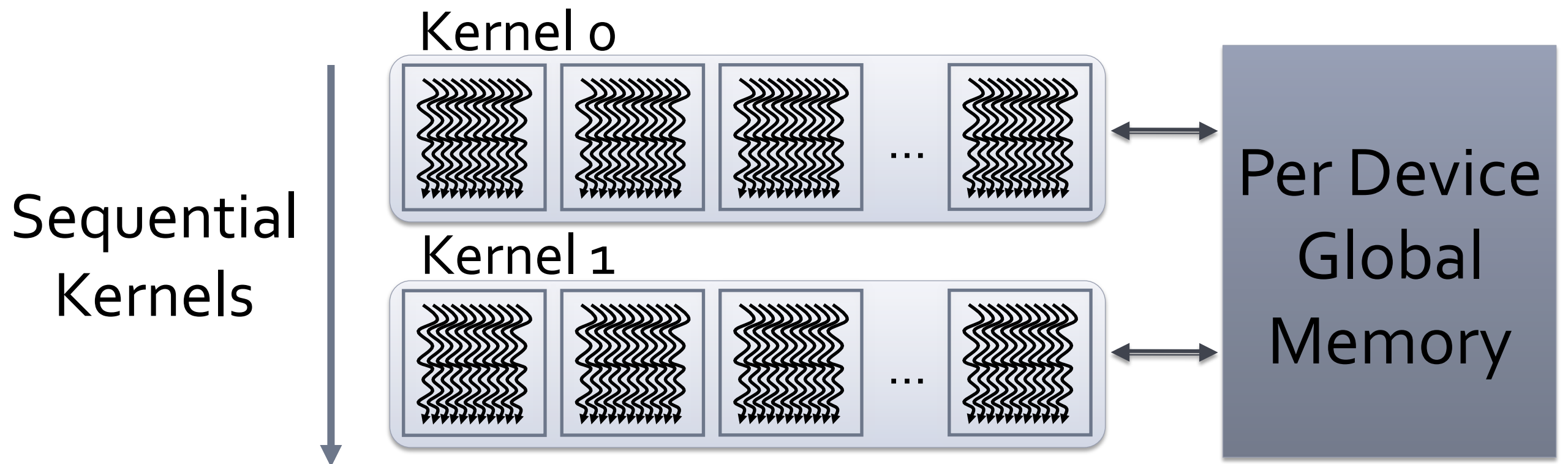- Thread blocks bring scalability!

Hello World: Vector Addition

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  //Run N/256 blocks of 256 threads each
  vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```
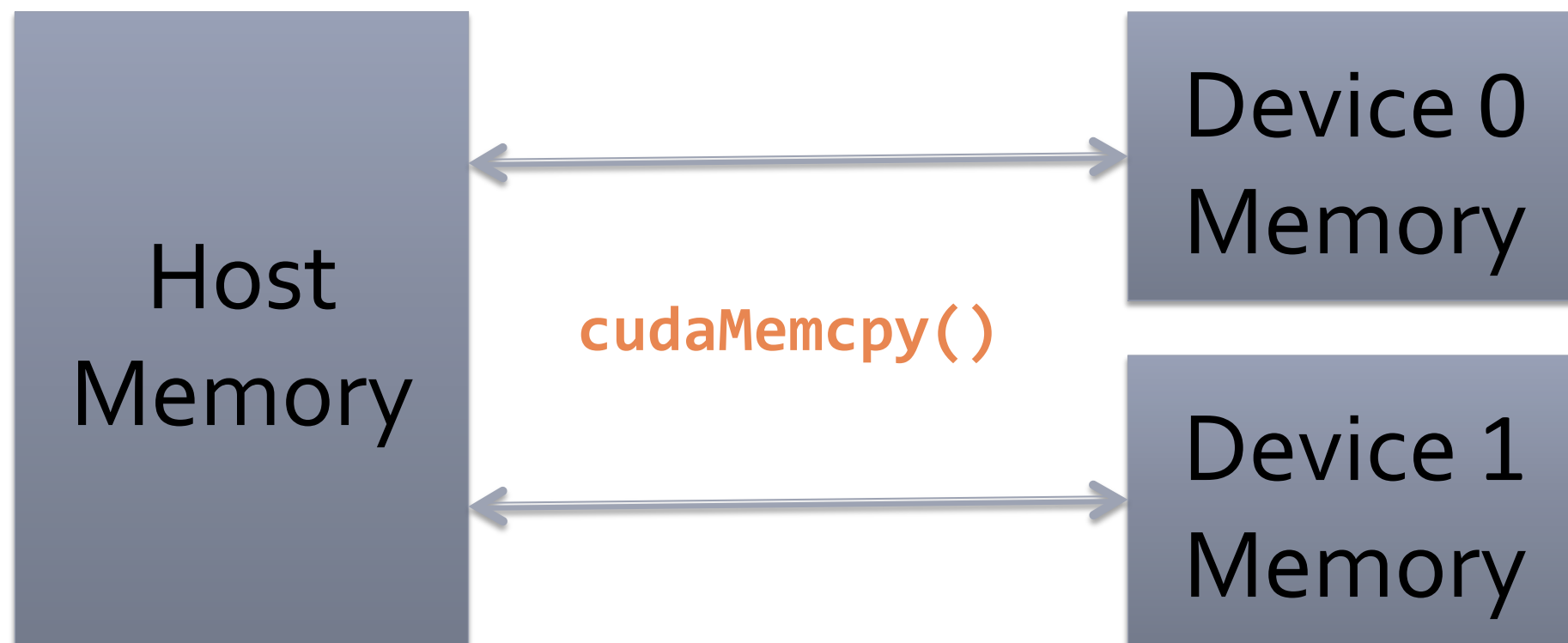
Memory model

Thread

Per-thread
Local Memory

Block

Per-block
Shared Memory

Memory model

Sequential Kernels

Kernel 0

...

Kernel 1

...

Per Device Global Memory

Memory model

Host Memory

cudaMemcpy()

Device 0 Memory

Device 1 Memory

Hello World: Managing Data

```c
int main() {
    int N = 256 * 1024;
    float* h_a = malloc(sizeof(float) * N);
    //Similarly for h_b, h_c. Initialize h_a, h_b

    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, sizeof(float) * N);
    //Similarly for d_b, d_c

    cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
    //Similarly for d_b

    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

    cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```

CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   //  kernel callable from host
__device__ void DeviceFunc(...);   //  function callable on device
__device__ int  GlobalVar;         //  variable in device memory
__shared__ int  SharedVar;         //  in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);      //  500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                    //  barrier synchronization
```

Using per-block shared memory

- Variables shared across block

  ```
  __shared__ int *begin, *end;
  ```

- Scratchpad memory

  ```
  __shared__ int scratch[BLOCKSIZE];
  scratch[threadIdx.x] = begin[threadIdx.x];
  // … compute on scratch values …
  begin[threadIdx.x] = scratch[threadIdx.x];
  ```

- Communicating values between threads

  ```
  scratch[threadIdx.x] = begin[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```

- Per-block shared memory is faster than L1 cache, slower than register file

- It is relatively small: register file is 2-4x larger
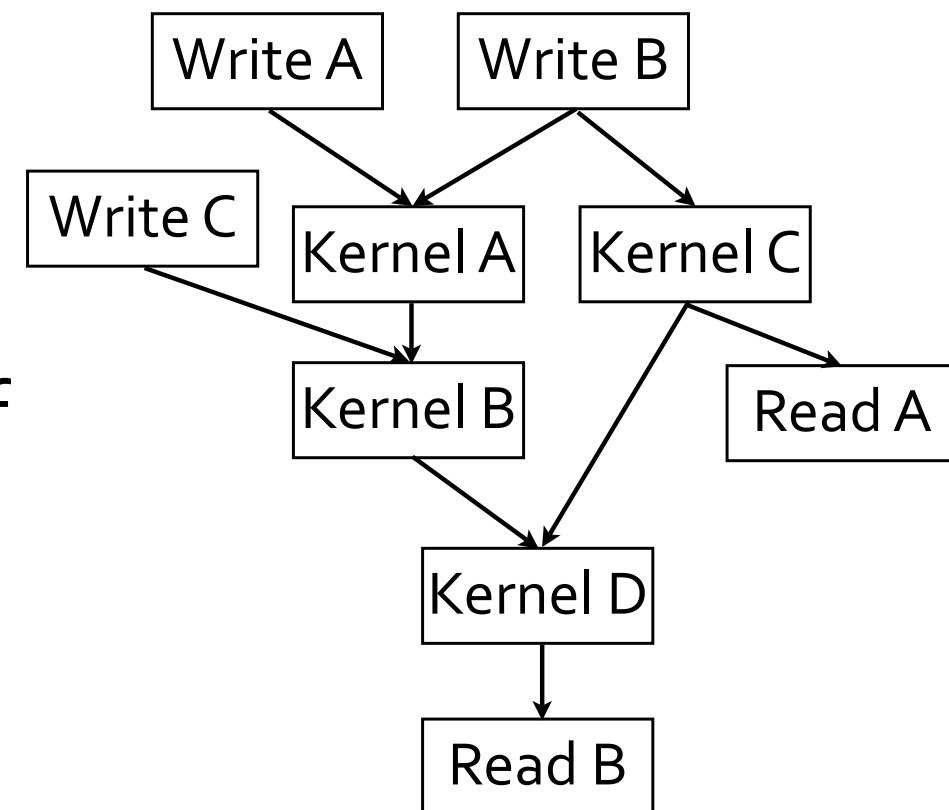
Block

Shared
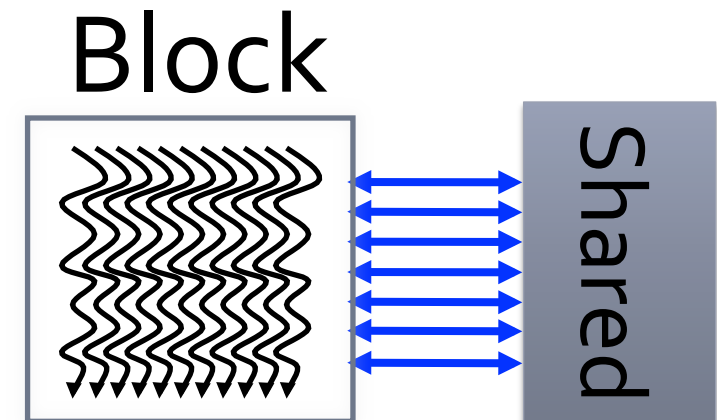
CUDA: Features available on GPU

- Double and single precision (IEEE compliant)

- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, …

- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, …

- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, …

OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/OMAP/etc.) are also on board
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
  - Runtime walks a dependence DAG of kernels/memory transfers

CUDA and OpenCL correspondence

- Thread   ⟷   Work-item
- Thread-block   ⟷   Work-group
- Global memory   ⟷   Global memory
- Constant memory   ⟷   Constant memory
- Shared memory   ⟷   Local memory
- Local memory   ⟷   Private memory
- __global__ function   ⟷   __kernel function
- __device__ function   ⟷   no qualification needed
- __constant__ variable   ⟷   __constant variable
- __device__ variable   ⟷   __global variable
- __shared__ variable   ⟷   __local variable

OpenCL and SIMD

- SIMD issues are handled separately by each runtime
- AMD GPU Runtime
  - Vectorize over 64-way SIMD, but not over 4/5-way VLIW
    - Use float4 vectors in your code
- AMD CPU Runtime
  - No vectorization
    - Use float4 vectors in your code (float8 when AVX appears?)
- Intel CPU Runtime
  - Vectorization optional, using float4/float8 vectors still good idea
- Nvidia GPU Runtime
  - Full vectorization, like CUDA
    - Prefers scalar code per work-item

Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000's of threads for full utilization

- Maximize on-chip work
  - on-chip memory orders of magnitude faster

- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps

- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line

Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.

- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs

- Threads: each thread is a SIMD vector lane

- Warps: A SIMD instruction acts on a "warp"
  - Warp width is 32 elements: *LOGICAL* SIMD width

- Thread blocks: Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per SM

Mapping CUDA to a GPU, *continued*

- The GPU is very deeply pipelined to maximize throughput

- This means that performance depends on the number of thread blocks which can be allocated on a processor

- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks

- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
  - For Fermi, target 20 registers or less per thread for full occupancy

Occupancy (Constants for Fermi)

- The Runtime tries to fit as many thread blocks simultaneously as possible on to an SM

  - The number of simultaneous thread blocks (B) is ≤ 8
- The number of warps per thread block (T) ≤ 32
- Each SM has scheduler space for 48 warps (W)

  - B * T ≤ W=48
- The number of threads per warp (V) is 32
- B * T * V * Registers per thread ≤ 32768
- B * Shared memory (bytes) per block ≤ 49152/16384

  - Depending on Shared memory/L1 cache configuration
- Occupancy is reported as B * T / W

SIMD & Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence

  - Write scalar SIMD code, the hardware schedules the SIMD execution

  - One caveat: __syncthreads() can't appear in a divergent path

    - This will cause programs to hang

  - Good performing code will try to keep the execution convergent within a warp

    - Warp divergence only costs because of a finite instruction cache

Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

  Kathy Yelick, Berkeley

Control	ALU	ALU
	ALU	ALU

Cache

DRAM

CPU

DRAM
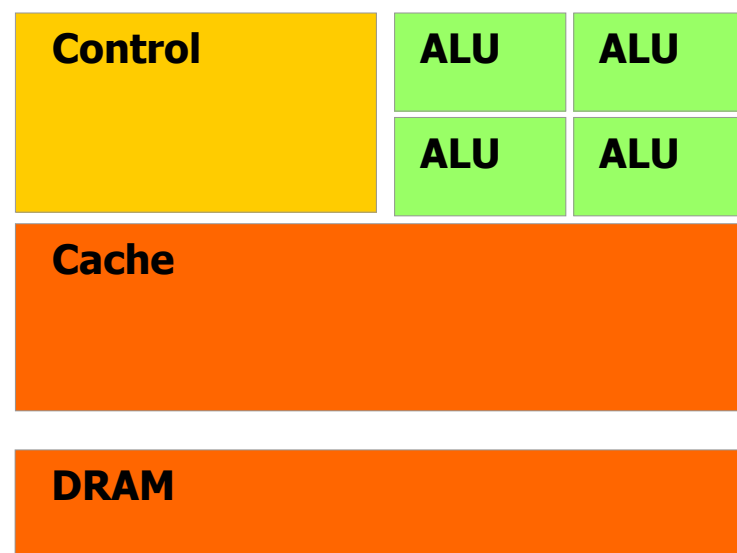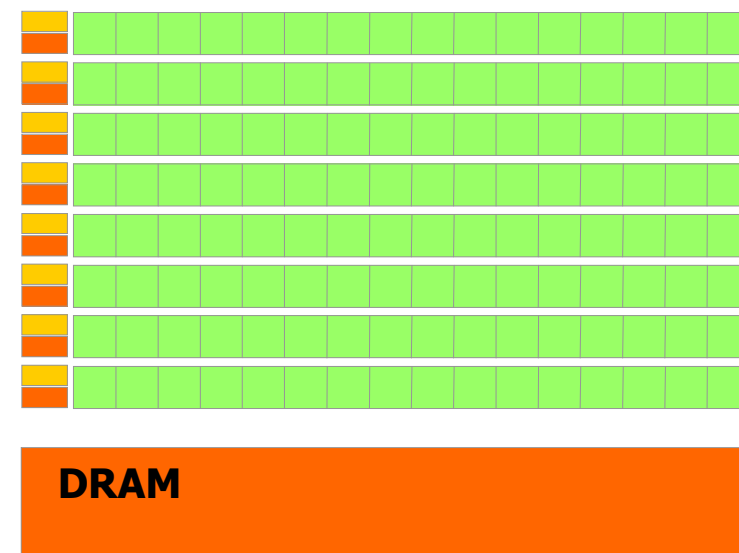
GPU

- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

- Virtually all processors have SIMD memory subsystems

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

cache line width
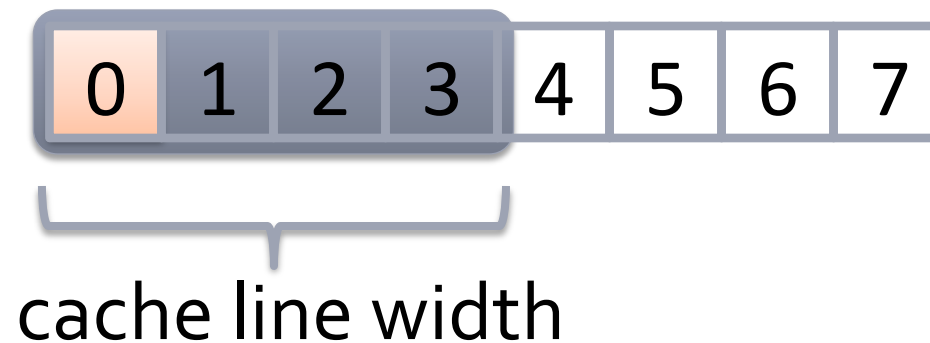
- This has two effects:
  - Sparse access wastes bandwidth

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

2 words used, 8 words loaded:
¼ effective bandwidth

  - Unaligned access wastes bandwidth

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
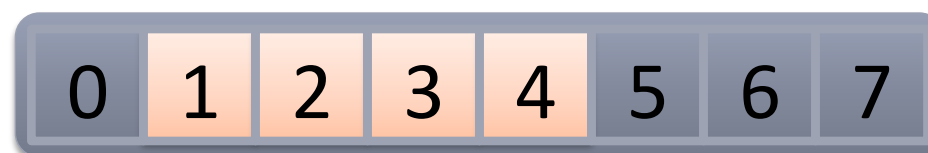
4 words used, 8 words loaded:
½ effective bandwidth

Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

Data Structure Padding

(row major)
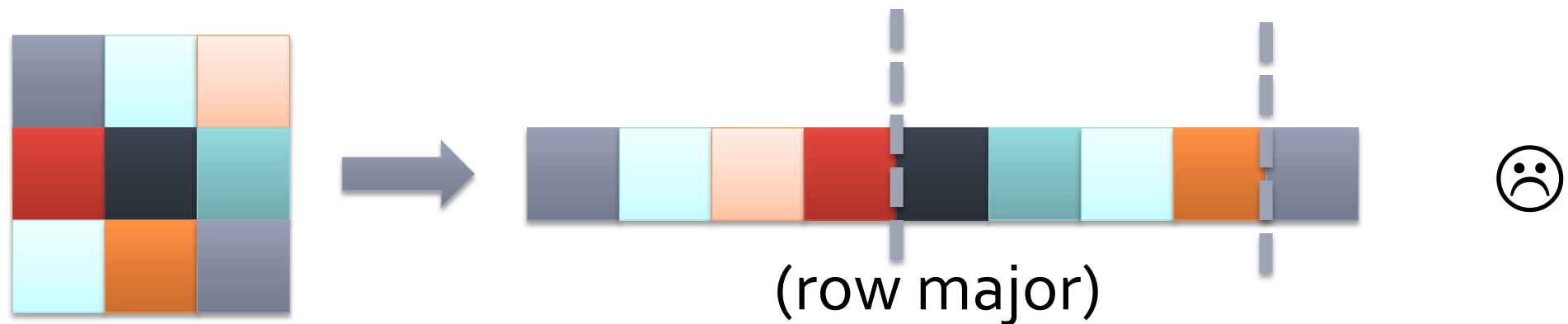
- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern
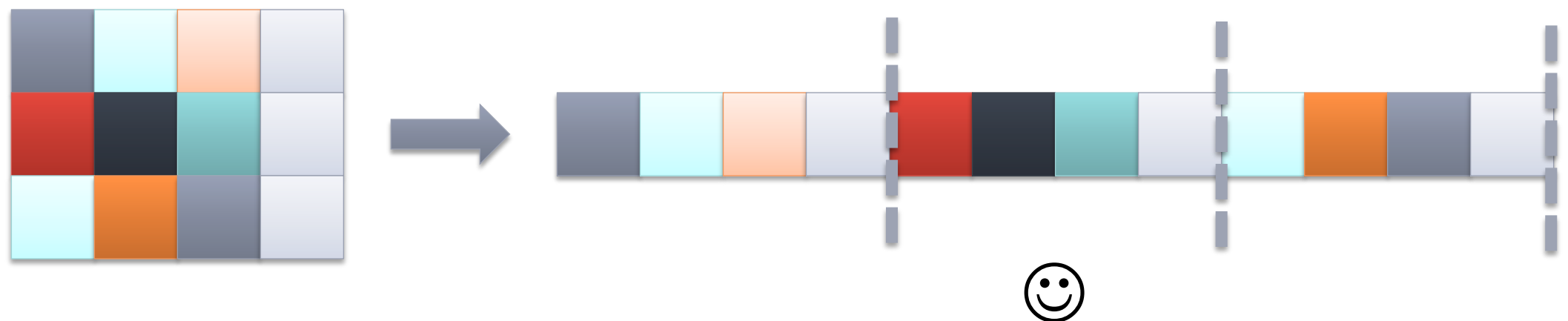
Sparse Matrix Vector Multiply
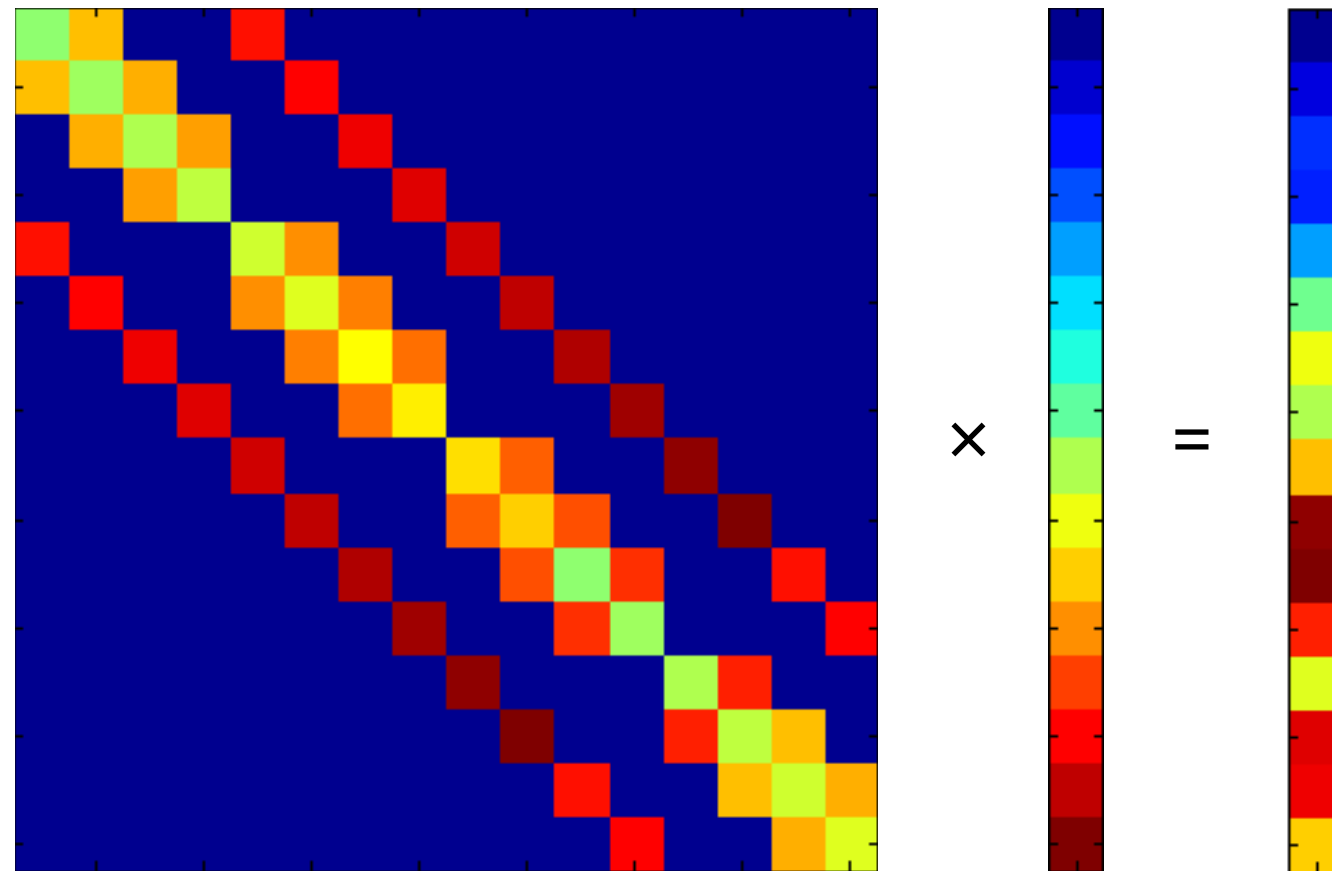
- Problem: Sparse Matrix Vector Multiplication
- How should we represent the matrix?
  - Can we take advantage of any structure in this matrix?

Diagonal representation
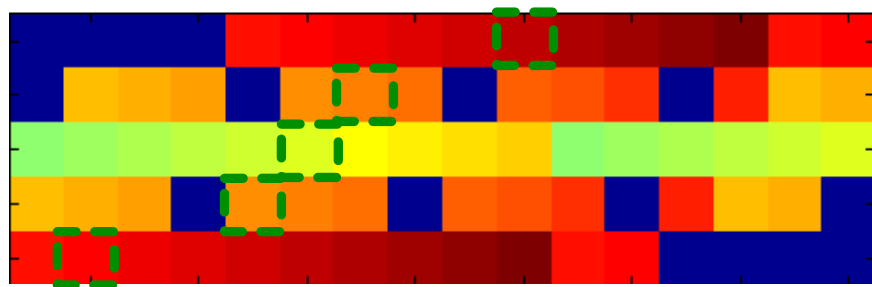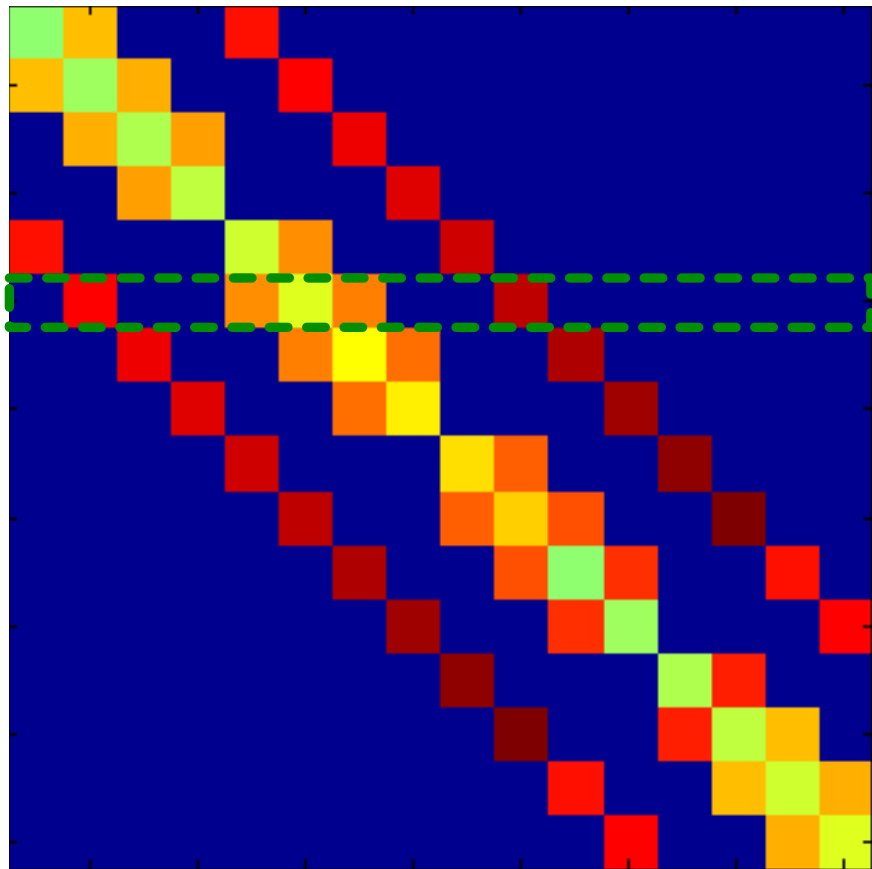
- Since this matrix has nonzeros only on diagonals, let's project the diagonals into vectors
- Sparse representation becomes dense
- Launch a thread per row
- Are we done?
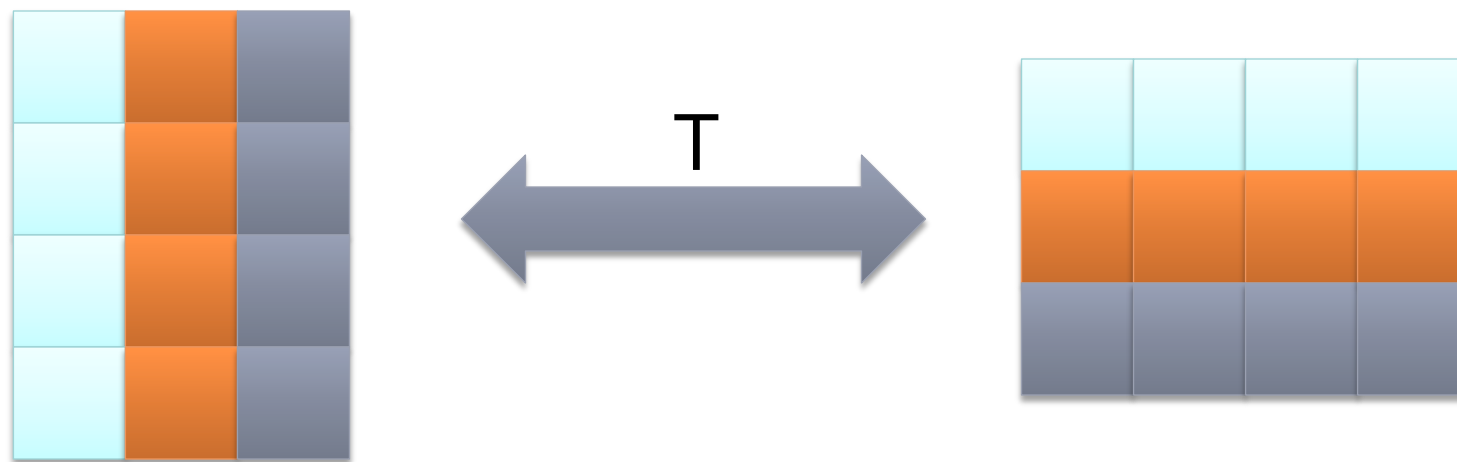
- The straightforward diagonal projection is not aligned

Optimized Diagonal Representation

padding

☺
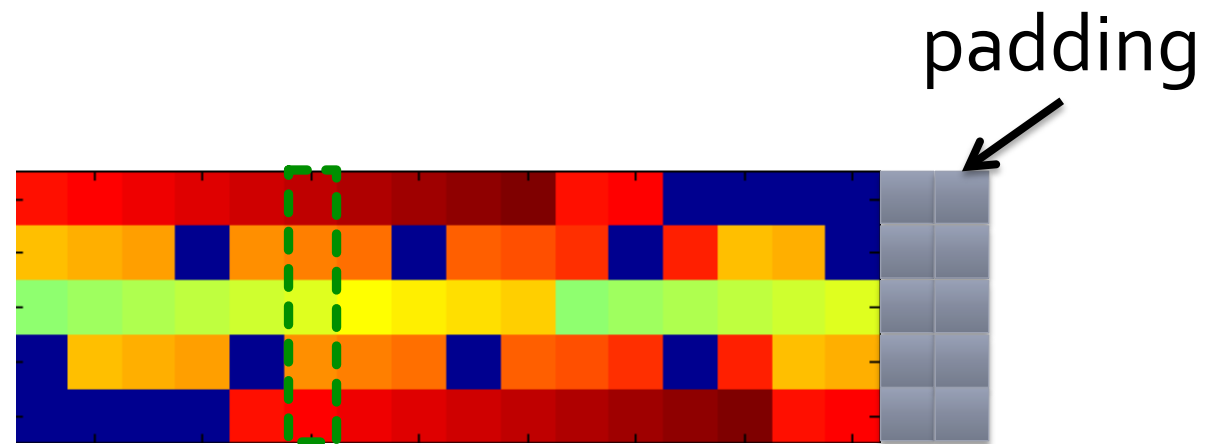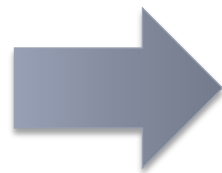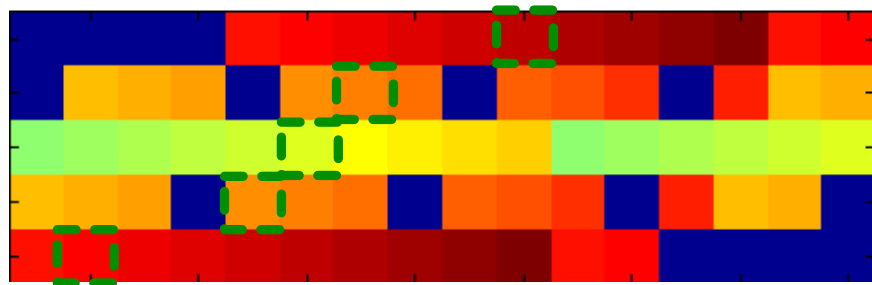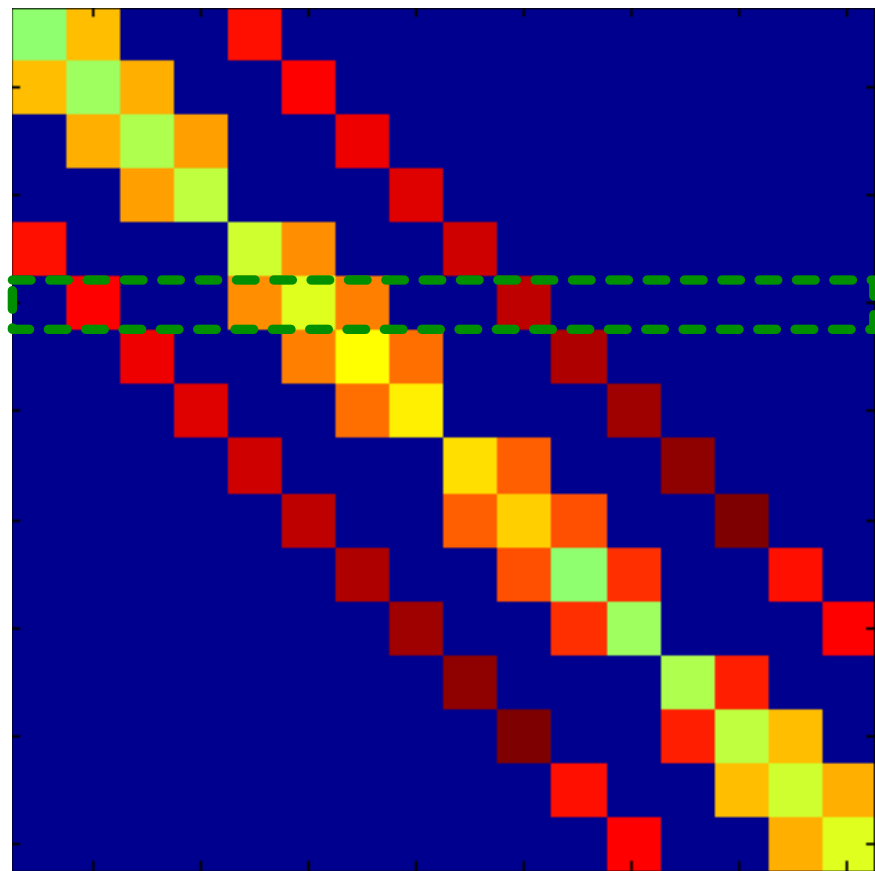
- Skew the diagonals again
- This ensures that all memory loads from matrix are coalesced
- Don't forget padding!

☹

SoA, AoS

- Different data access patterns may also require transposing data structures
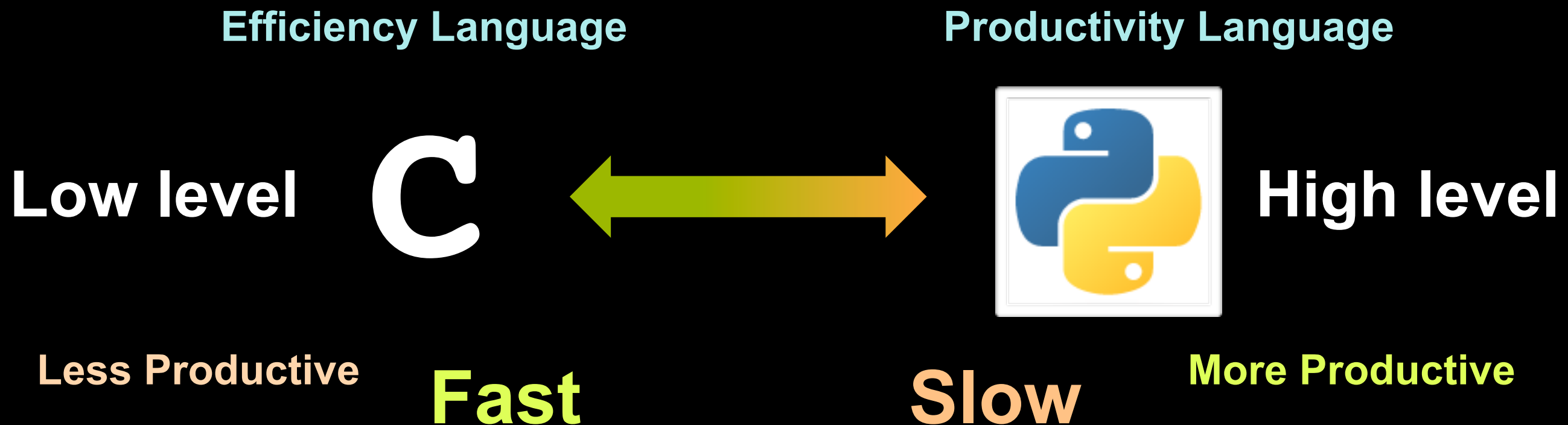

Array of Structs   T   Structure of Arrays

- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses
- Use shared memory to handle block transposes

Certainties: Death and Taxes

- Productivity is often in tension with efficiency
  - This is often called the "abstraction tax"

Efficiency Language **Productivity Language**

Low level C ⟷ **High level**

Less Productive   **Fast**   **Slow**   **More Productive**

The Concrete Tax

- Parallel programming also gives us a "concrete tax"
  - How many of you have tried to write … which is faster than a vendor supplied library?
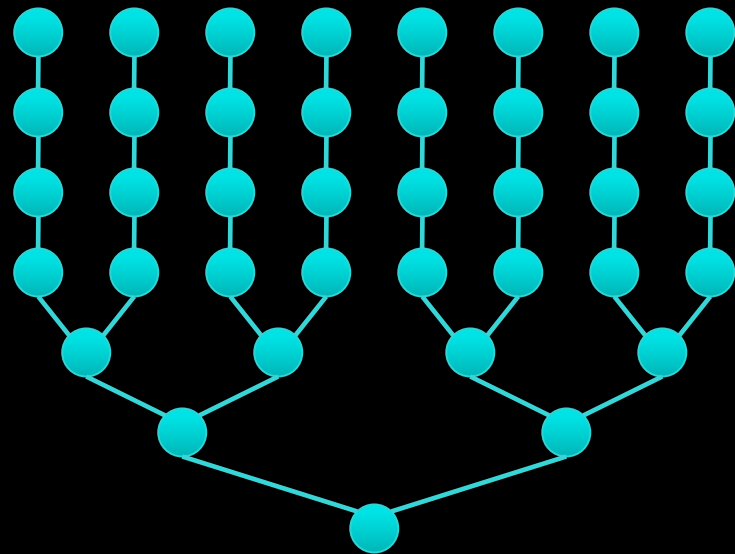
**FFT** **Sort** **Reduce**
SGEMM **Scan**

- Divergent Parallel Architectures means performance portability is increasingly elusive

- Low-level programming models tie you to a particular piece of hardware

- And if you're like me, often make your code slow
  - My SGEMM isn't as good as NVIDIA's

The Concrete Tax: A Case Study

Wide Parallel
Reduction
(good for GPU)

Narrow Parallel
Reduction
(good for CPU)
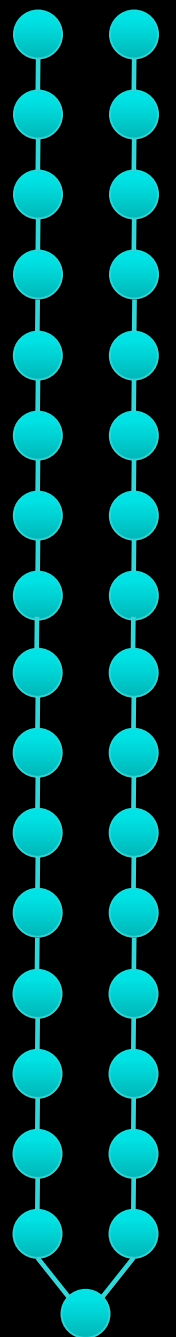
- OpenCL experiment on CPU and GPU

- Two optimized reductions, one for CPU, one for GPU

- Running GPU code on CPU:

  - 40X performance loss compared to CPU optimized code

- Running CPU on GPU:

  - ~100X performance loss compared to GPU optimized code

- Concrete code led to overspecialization

Abstraction, *cont.*

- Reduction is one of the simplest parallel computations

- Performance differentials are even starker as complexity increases

- There's a need for abstractions at many levels
  - Primitive computations (BLAS, Data-parallel primitives)
  - Domain-specific languages

- These abstractions make parallel programming more efficient *and* more productive

- A C++ template library for CUDA
  - Mimics the C++ STL

- Containers
  - On host and device

- Algorithms
  - Sorting, reduction, scan, etc.

Diving In

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Objectives

- Programmer productivity
  - Build complex applications quickly

- Encourage generic programming
  - Leverage parallel primitives

- High performance
  - Efficient mapping to hardware

Containers

- Concise and readable code
  - Avoids common memory management errors

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] <<
std::endl;
```

Iterators

- Pair of iterators defines a *range*

```
// allocate device memory
device_vector<int> d_vec(10);

// declare iterator variables
device_vector<int>::iterator begin   =
d_vec.begin();
device_vector<int>::iterator end     = d_vec.end();
device_vector<int>::iterator middle = begin + 5;

// sum first and second halves
int sum_half1 = reduce(begin, middle);
int sum_half2 = reduce(middle, end);

// empty range
int empty = reduce(begin, begin);
```

Iterators

- Iterators act like pointers

```
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end   = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```

Iterators

- Encode memory location
  - Automatic algorithm selection

```cpp
// initialize random values on host
host_vector<int> h_vec(100);
generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = reduce(d_vec.begin(), d_vec.end());
```

Algorithms

- Elementwise operations
  - `for_each`, `transform`, `gather`, `scatter` …
- Reductions
  - `reduce`, `inner_product`, `reduce_by_key` …
- Prefix-Sums
  - `inclusive_scan`, `inclusive_scan_by_key` …
- Sorting
  - `sort`, `stable_sort`, `sort_by_key` …

Algorithms

- Standard operators

```cpp
// allocate memory
device_vector<int>  A(10);
device_vector<int>  B(10);
device_vector<int>  C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```

Algorithms

Standard data types

```cpp
// allocate device memory
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(),
f_vec.end());
```

Custom Types & Operators

```cpp
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};


// declare storage
device_vector<float2> input  = ...
device_vector<float2> output = ...

// create function object or 'functor'
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```

Custom Types & Operators

```cpp
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

Interoperability

- Convert iterators to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// Note: ptr cannot be dereferenced on the host!
```

Recap

- Containers manage memory
  - Help avoid common errors

- Iterators define ranges
  - Know where data lives

- Algorithms act on ranges
  - Support general types and operators

Explicit versus implicit parallelism

- CUDA is explicit
  - Programmer's responsibility to schedule resources
  - Decompose algorithm into kernels
  - Decompose kernels into blocks
  - Decompose blocks into threads

Kernel 1
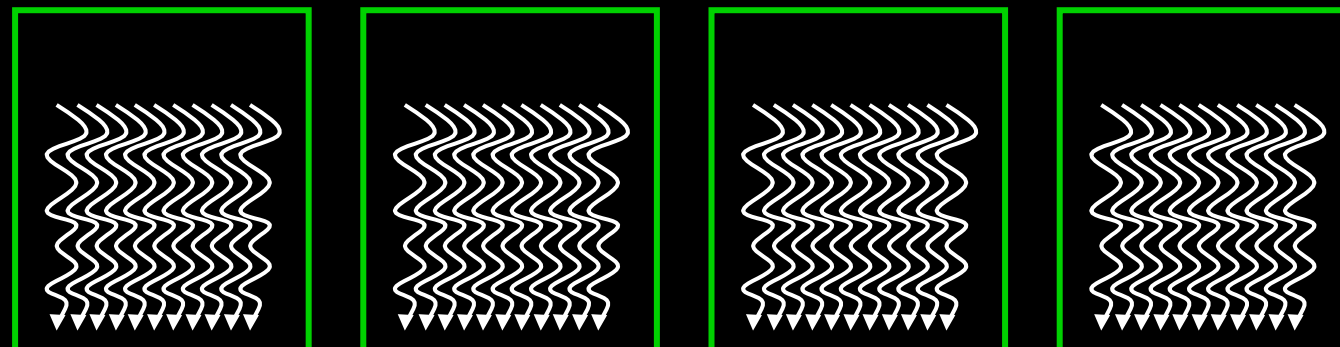
Kernel 2
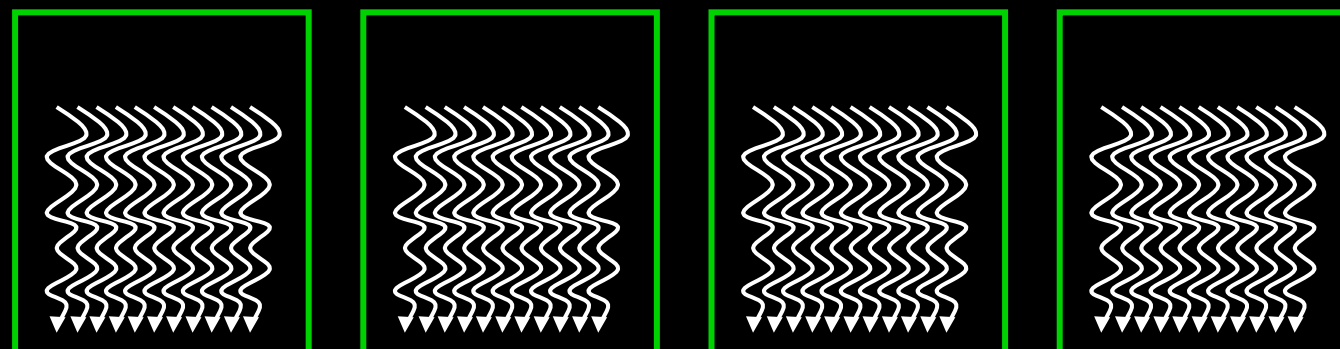
Explicit versus implicit parallelism

- SAXPY in CUDA

```
__global__
void SAXPY(int n, float a, float * x, float * y)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < n)
        y[i] = a * x[i] + y[i];
}

SAXPY <<< n/256, 256 >>>(n, a, x, y);
```

Explicit versus implicit parallelism

- SAXPY in CUDA

```
__global__
void SAXPY(int n, float a, float * x, float * y)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < n)
        y[i] = a * x[i] + y[i];
}


SAXPY <<< n/256, 256 >>>(n, a, x, y);
```

Decomposition

Explicit versus implicit parallelism

- SAXPY in Thrust

```cpp
// C++ functor replaces __global__ function
struct saxpy
{
    float a;

    saxpy(float _a) : a(_a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

transform(x.begin(), x.end(), y.begin(), y.begin(),
    saxpy(a));
```

Implicitly Parallel

- Algorithms expose lots of *fine-grained* parallelism
  - Generally expose O(N) independent threads of execution
  - Minimal constraints on implementation details

- Programmer identifies opportunities for parallelism
  - Thrust determines explicit decomposition onto hardware

- Finding parallelism in sequential code is hard
  - Mapping parallel computations onto hardware is easier

Productivity Implications

- Consider a serial reduction

```
// sum reduction
int sum = 0;
for(i = 0; i < n; ++i)
    sum += v[i];
```

Productivity Implications

- Consider a serial reduction

```
// product reduction
int product = 1;
for(i = 0; i < n; ++i)
  product *= v[i];
```

Productivity Implications

- Consider a serial reduction

```cpp
// max reduction
int max = 0;
for(i = 0; i < n; ++i)
  max = std::max(max,v[i]);
```

Productivity Implications

- Compare to low-level CUDA

```
int sum = 0;
for(i = 0; i < n; ++i)
  sum += v[i];
```

```
__global__
void block_sum(const float *input,
               float *per_block_results,
               const size_t n)
{
  extern __shared__ float sdata[];

  unsigned int i = blockIdx.x *
   blockDim.x + threadIdx.x;

  // load input into __shared__ memory
  float x = 0;
  if(i < n)
  {
    x = input[i];

                  ...
```

Leveraging Parallel Primitives

- Use `sort` liberally

data type	std::sort	tbb::parallel_sort	thrust::sort
char	25.1	68.3	3532.2
short	15.1	46.8	1741.6
int	10.6	35.1	804.8
long	10.3	34.5	291.4
float	8.7	28.4	819.8
double	8.5	28.2	358.9

Intel Core i7 950   NVIDIA GeForce 480

Input-Sensitive Optimizations
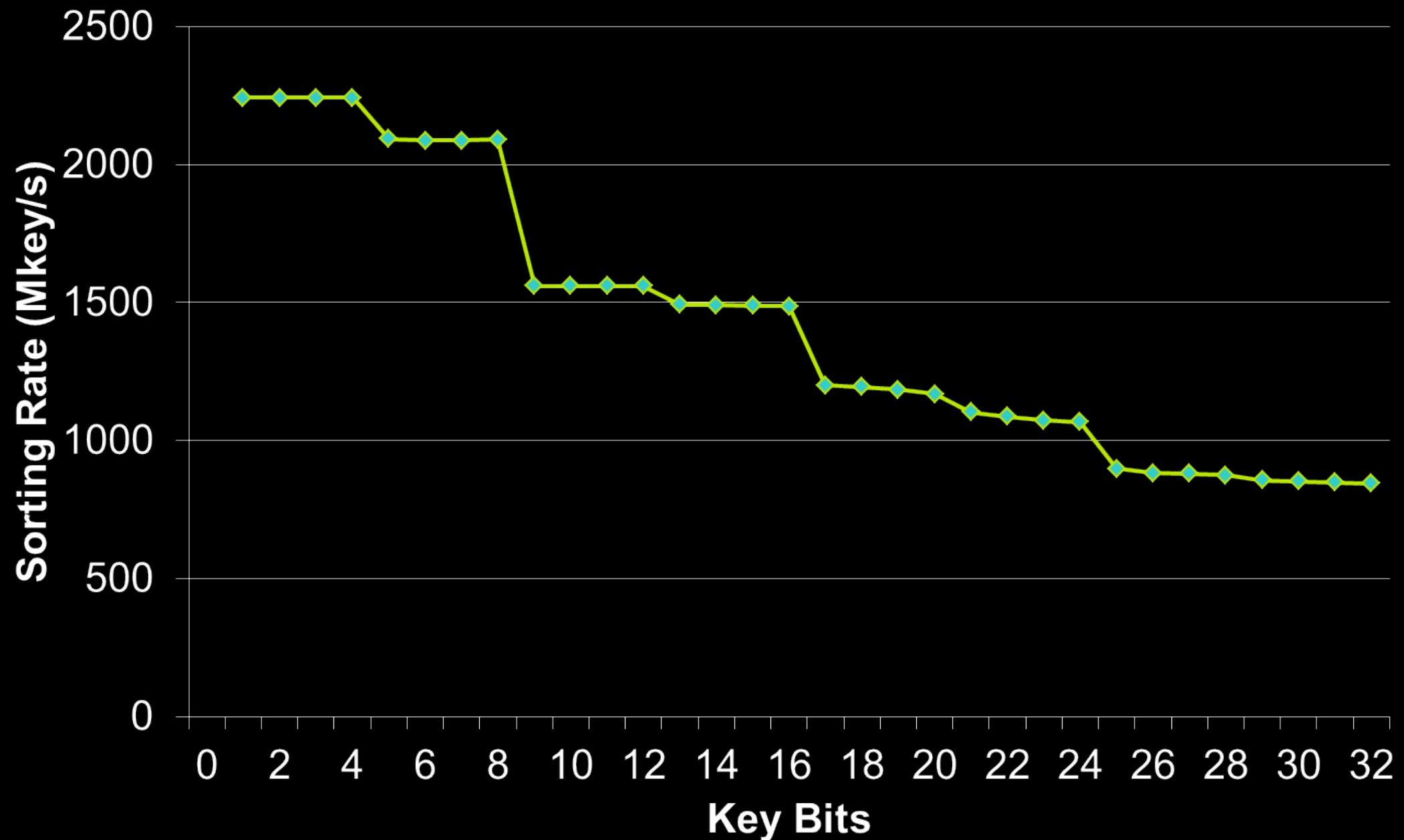
Leveraging Parallel Primitives

- Combine `sort` with `reduce_by_key`
  - Keyed reduction
  - Bring like items together, collapse
  - Poor man's MapReduce

- Can often be faster than custom solutions
  - I wrote an image histogram routine in CUDA
  - Bit-level optimizations and shared memory atomics
  - Was 2x slower than `thrust::sort` + `thrust::reduce_by_key`

Thrust on Google Code

- Quick Start Guide

- Examples

- Documentation

- Mailing list (thrust-users)

Summary

- Manycore processors provide useful parallelism
- Programming models like CUDA and OpenCL enable productive parallel programming
- They abstract SIMD, making it easy to use wide SIMD vectors
- CUDA and OpenCL encourages SIMD friendly, highly scalable algorithm design and implementation
- Thrust is a productive C++ library for CUDA development

Questions?

Bryan Catanzaro

bcatanzaro@nvidia.com

http://research.nvidia.com