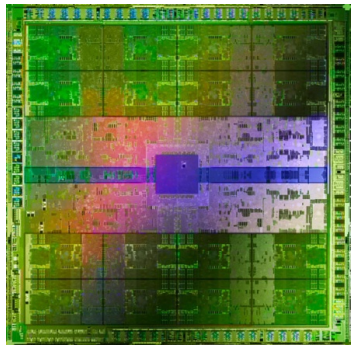
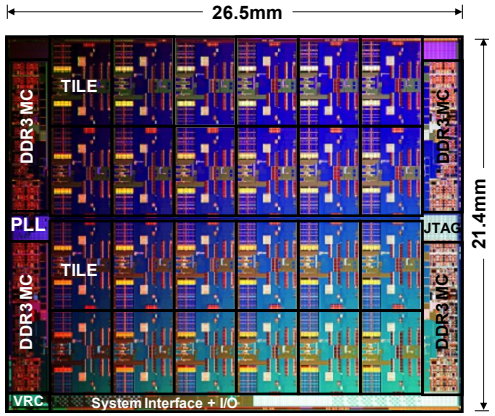
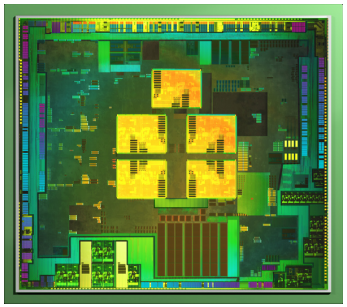


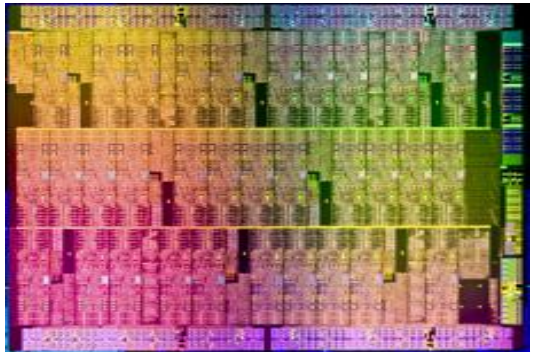
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



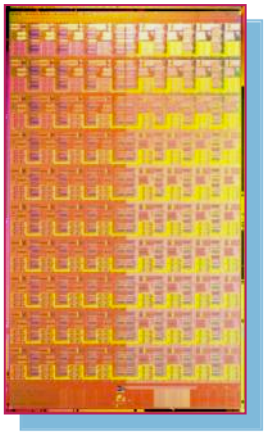
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



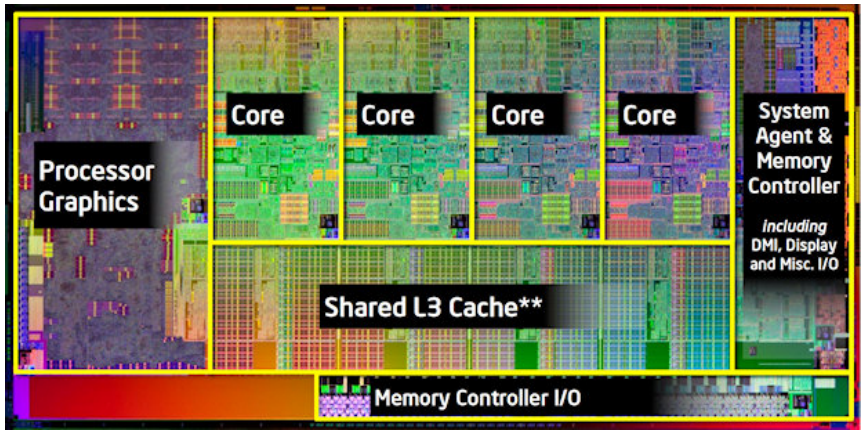
An Intel MIC processor

Hands-on Intro to CUDA for OpenCL programmers

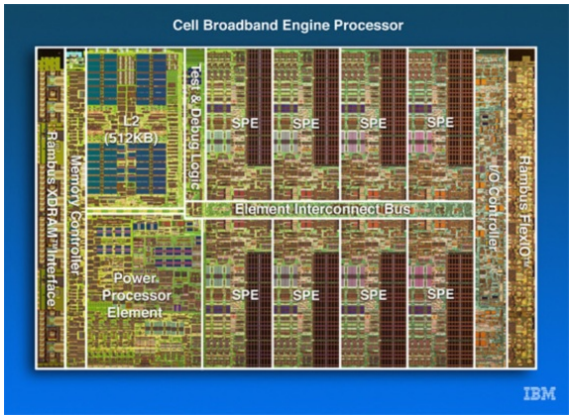
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



IBM Cell Broadband engine processor

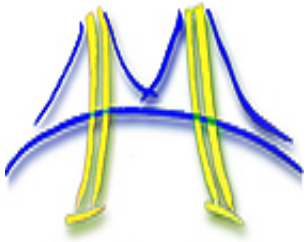
Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

Third party names are the property of their owners

Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault … don't blame our collaborators.

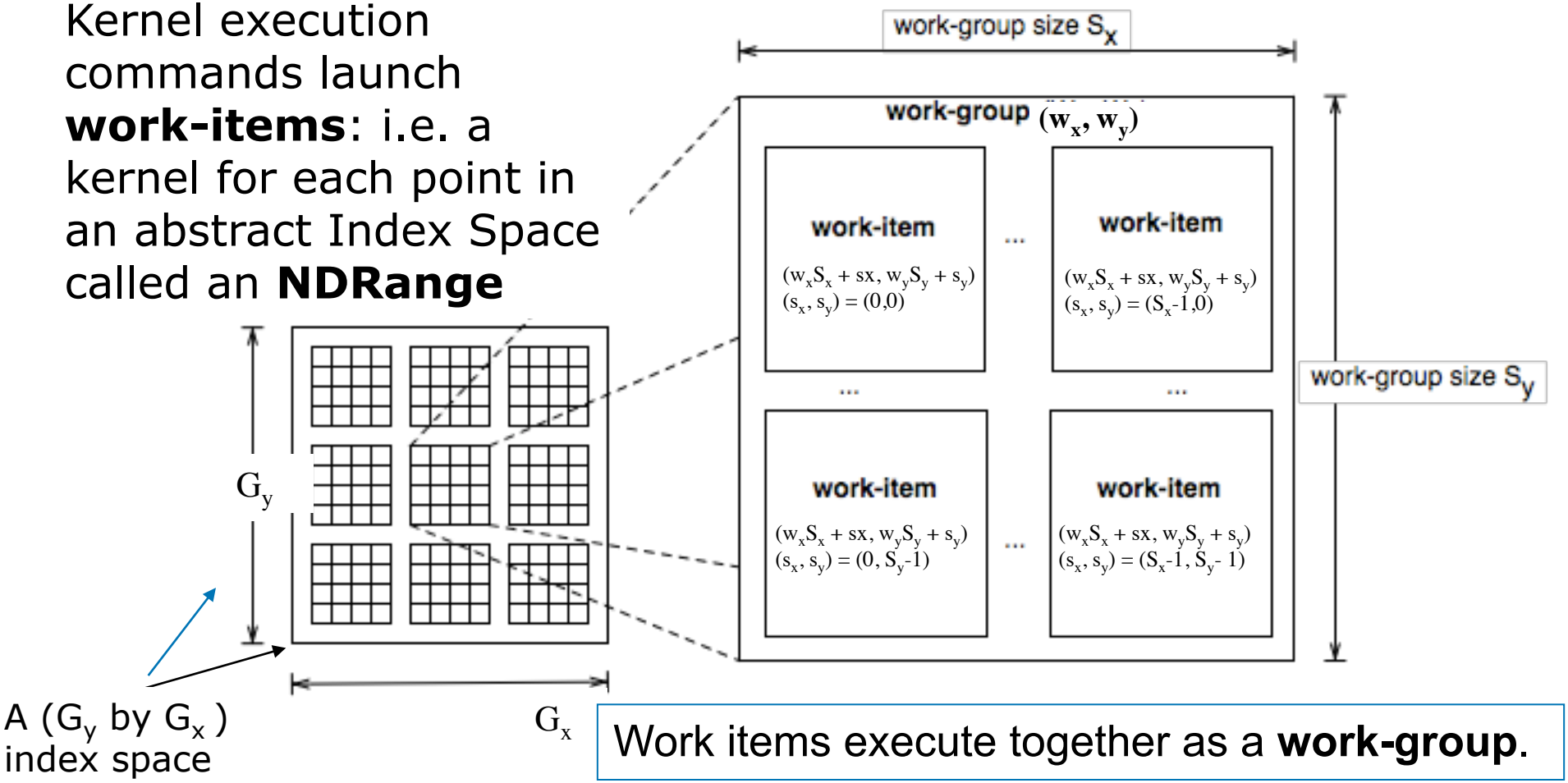


Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

Recall the OpenCL Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



Work items execute together as a **work-group**.

OpenCL vs. CUDA Terminology

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

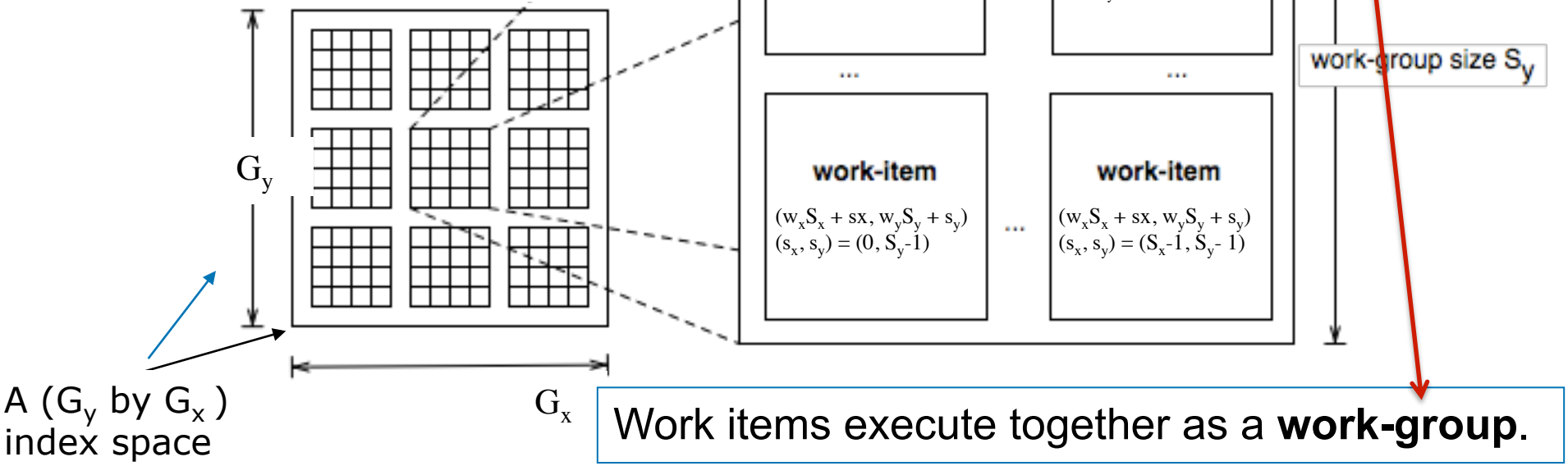
CUDA Stream

Thread Block

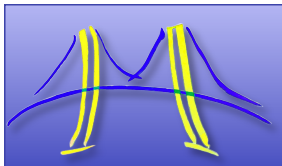
Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Threads

Grid



Work items execute together as a **work-group**.



What is a CUDA thread?

- Logically, each CUDA thread is its own very lightweight **independent execution context**
  - Has its own control flow and PC, register file, call stack, ...
  - Can access any GPU memory address at any time
  - Identifiable uniquely within a grid by the six integers: `threadIdx.{x,y,z}, blockIdx.{x,y,z}`
- **Very fine granularity**: do not expect any single thread to do a substantial fraction of an expensive computation
  - At full occupancy, each Thread has 21 32-bit registers
  - ... 1,536 Threads share a 48 KB L1 Cache / `__shared__` mem

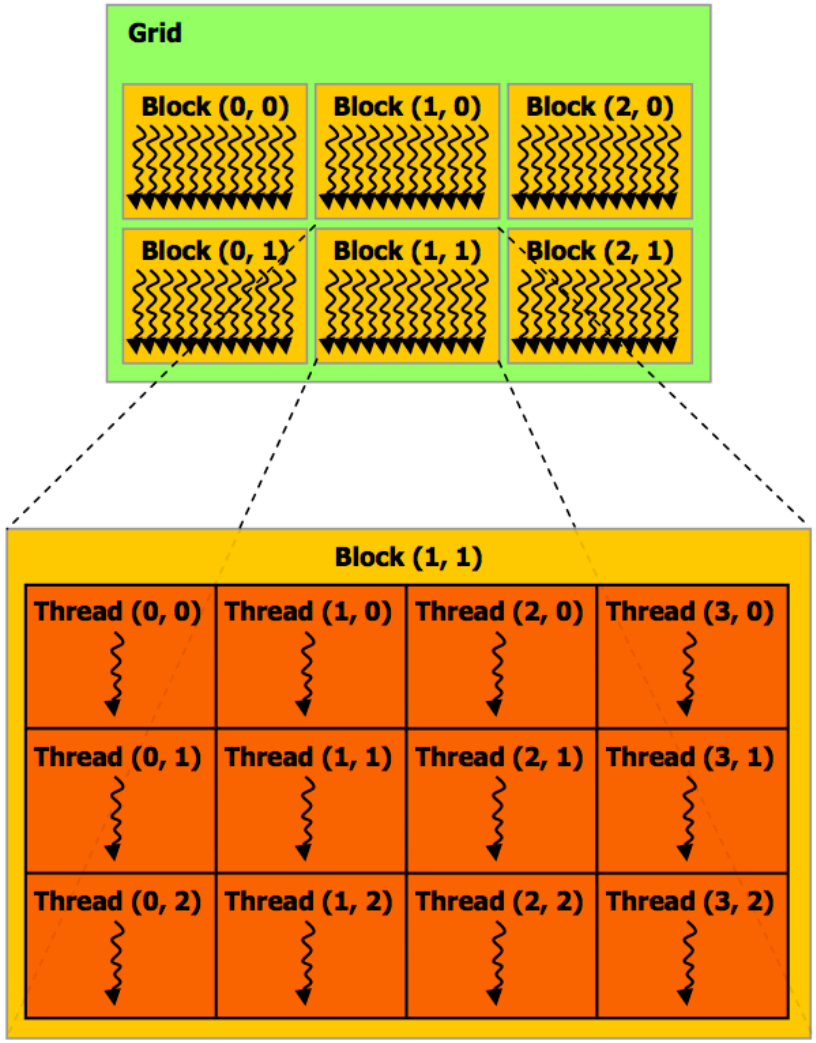
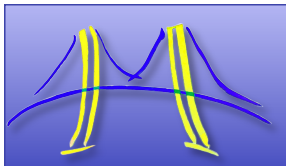
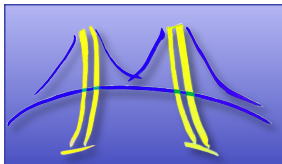


Image source: The CUDA programming guide



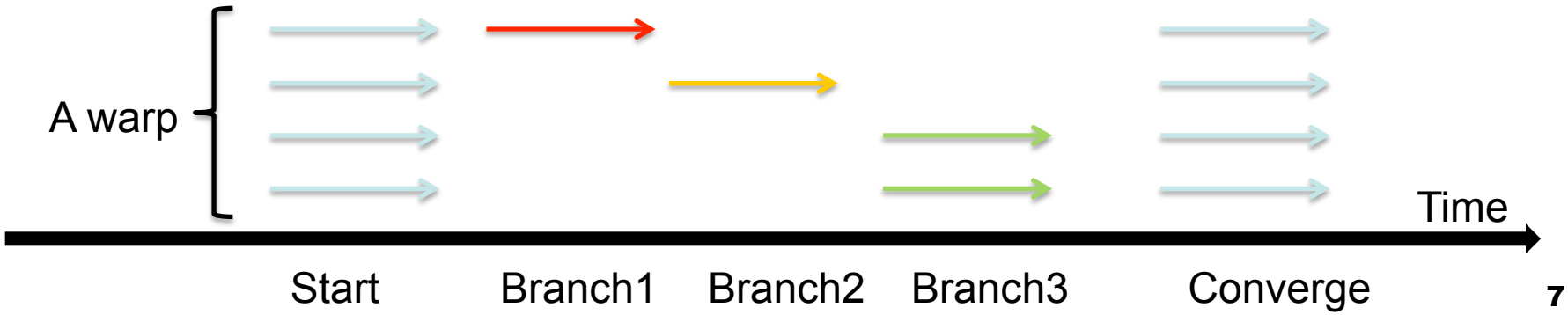
What is a CUDA warp?

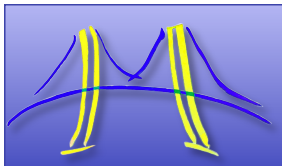
- A group of 32 CUDA threads that execute simultaneously
  - Execution hardware is most efficiently utilized when all threads in a warp execute instructions from the same PC.
  - Identifiable uniquely by dividing the Thread Index by 32
  - If threads in a warp ***diverge*** (execute different PCs), then some execution pipelines go unused
  - If threads in a warp access aligned, contiguous blocks of DRAM, the accesses are ***coalesced*** into a single high-bandwidth access
- The minimum granularity of efficient SIMD execution, and the maximum hardware SIMD width in a CUDA processor



Single Instruction Multiple Data

- Individual threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Threads not following the current branch will be disabled





What is a CUDA thread block?

- A thread block is a **virtualized multi-threaded core**
  - Configured at kernel-launch to have a number of scalar processors, registers, `__shared__` memory
  - Consists of a number (32-1024) of CUDA threads, who all share the integer identifier `blockIdx.{x,y,z}`
- ... executing a **data parallel task** of moderate granularity
  - The cacheable working-set should fit into the register file and the L1 cache
  - All threads in a block share a (small) instruction cache and synchronize via the barrier intrinsic `__syncthreads()`

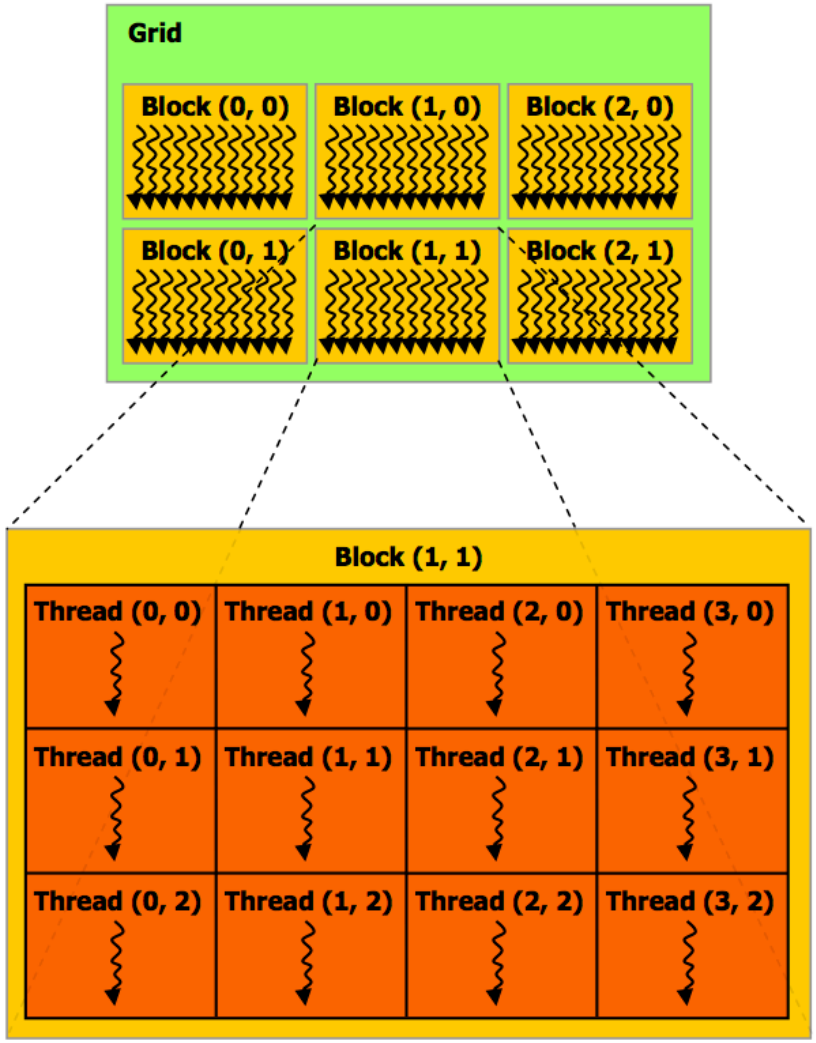
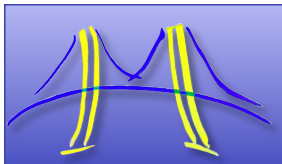


Image source: The CUDA programming guide



What is a CUDA grid?

- A set of Thread Blocks performing related computations
  - All threads in a single kernel call have the same entry point and function arguments, initially differing only in `blockIdx.{x,y,z}`
  - Thread blocks in a grid may execute any code they want, e.g. `switch (blockIdx.x) { ... }` incurs no penalty
- There is an implicit global barrier between kernel calls
- Thread blocks of a kernel call must be **parallel sub-tasks**
  - Program must be valid for ***any interleaving*** of block executions
  - The flexibility of the memory system technically allows Thread Blocks to communicate and synchronize in arbitrary ways ...
  - But there is **no guarantee** that all Thread Blocks execute concurrently, and inter-block communication is risky!

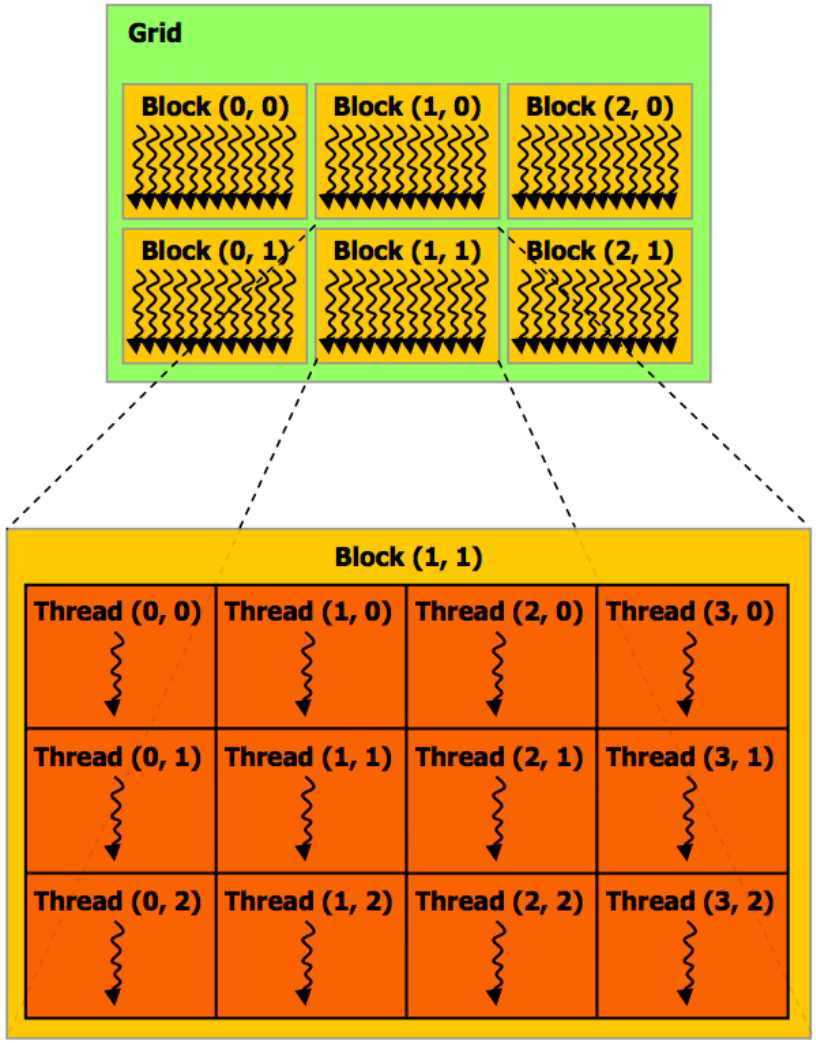
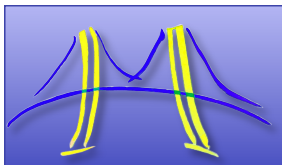
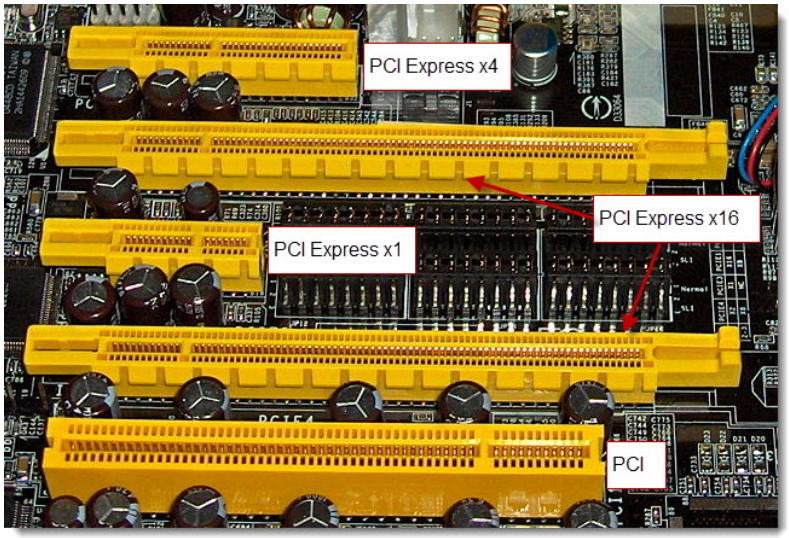


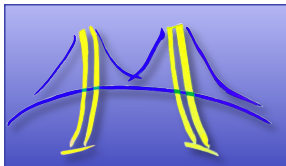
Image source: The CUDA programming guide



CUDA Host Runtime Support

- CUDA is a heterogeneous programming model
  - Sequential code runs in the “Host Thread” on a CPU core, and the “Device” code runs on the many cores of the GPU
  - The Host and the Device communicate via a PCI-Express link
  - The PCI-E link is slow (high latency, low bandwidth)
    - Desirable to minimize the amount of data transferred and the number of transfers





CUDA Host Runtime Support

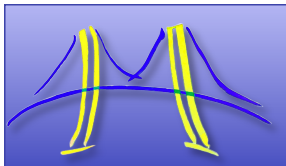
- Allocation/Deallocation of memory on the GPU:
  - **cudaMalloc(void**, int), cudaFree(void*)**
- Memory transfers to/from the GPU:
  - **cudaMemcpy(void*,void*,int, dir)**
  - **dir can be "cudaMemcpyHostToDevice" or "cudaMemcpyDeviceToHost"**

```
int main () {
    int N = (1024*1024);
 // pointers to array on the CPU
    float *h_a = new float[N];
    for(int i=0; i < N; i++) h_a[i] = i;
 // pointers to array on the GPU
    float *g_a;
    cudaMalloc(&g_a, sizeof(float)*N);
    cudaMemcpy(g_a, h_a, sizeof(float)*N,
        cudaMemcpyHostToDevice);
```

Create an array on the host CPU and fill with data

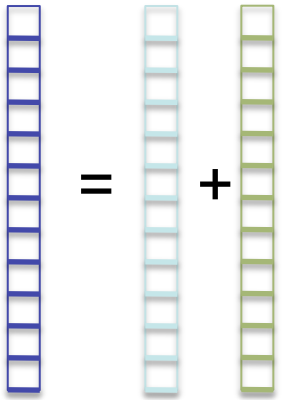
Allocate array on the GPU

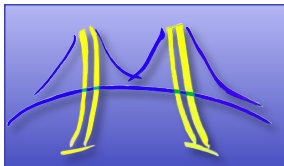
Copy data from host CPU upto the GPU



Hello World: Vector Addition (C++)

```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    d_a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (d_a, d_b, d_c, N);
}
```

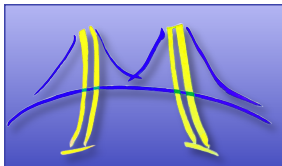




Hello World: Vector Addition (CUDA)

```
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}
int main () {
    int N = ... ;    float *d_a, *d_b, *d_c;
    std::vecor<float> h_a[LENGTH]; // host side data
    std::vecor<float> h_c[LENGTH]; // host side result vec
   // Selecte default device
    cudaSetDevice(0);
   // ... allocate arrays, fill with data, copy data to device
    cudaMalloc (&d_a,  sizeof(float) * N);   cudaMemcpy(d_a, &h_a[0], sizeof
   (float)*LENGTH, cudaMemcpyHostToDevice);
   // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
   // ... copy data back to host
    cudaMemcpy(&h_c[0], d_c, sizeof(float)*LENGTH, cudaMemcpyDeviceToHost);
    cudaFree(d_a);
}
```

Note: this is a partial solution. We don't show all the data allocation and copies for each array (just a and c)



Hello World: Vector Addition (CUDA)

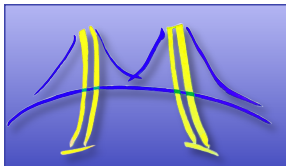
You should test that i<N in case you have extra threads when block size doesn't evenly divide N

```
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}
int main () {
    int N = ... ;    float *d_a, *d_b, *d_c;
    std::vecor<float> h_a[LENGTH]; // host side
    std::vecor<float> h_c[LENGTH]; // host side
   // Selecte default device
    cudaSetDevice(0);
   // ... allocate arrays, fill with data, copy
    cudaMalloc (&d_a,  sizeof(float) * N);   cu
   (float)*LENGTH, cudaMemcpyHostToDevice);
   // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
   // ... copy data back to host
    cudaMemc
    cudaFree(c
}
```

Note: this is a partial solution. We don't show all the data allocation and

When you launch a kernel, you must specify two parameters of type dim3. The first specifies the global dimension of the grid (one to three dimensions) and the second species the size of a "thread block".

(N+255)/256) assures that you round up on the integer division and have enough blocks even when N isn't evenly divided by the block size.



Vector addition: side by side

```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    d_a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (d_a, d_b, d_c, N);
}
```

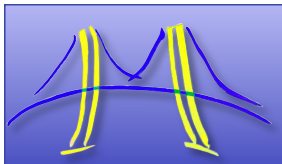
```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    cudaMalloc (&d_a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c,
    N);
}
```

CUDA Exercise 1

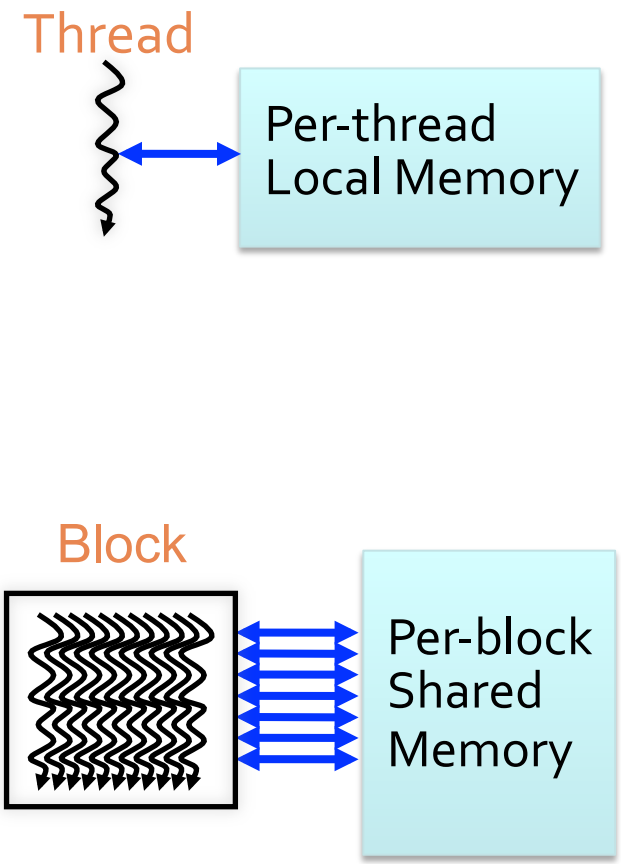
- Goal
  - Verify that you really understand the constructs by playing with the vector add program
- Problem
  - Start with the vector addition program we provide, create a CUDA version of the program.
- Extra work
  - Experiment with different Grid sizes. Find grid sizes that lead to the best performance. Relate what you observe to the size of a CUDA Warp.

Kernel example: void __global__ vfunc(const float *a, float *c, const int N);

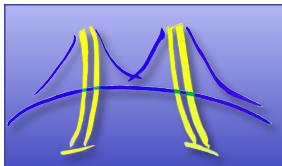
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
cudaSetDevice(0);
cudaMalloc (&a,  sizeof(float) * LEN);
cudaMemcpy(a,&a[0],sizeof(float)*LEN,cudaMemcpyHostToDevice);
vfunc<<<Glob_size,Block_size>>>(a, c, LEN);
cudaMemcpy(&c[0],c, sizeof(float)*LEN,cudaMemcpyDeviceToHost);
cudaFree(a);
```



CUDA memory hierarchy

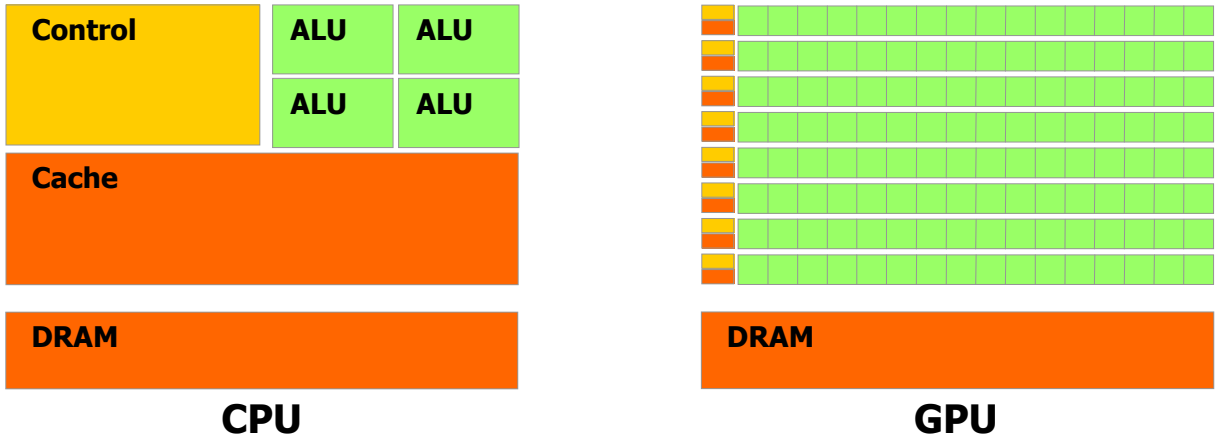


- Each CUDA thread has private access to a configurable number of registers
  - The 64 KB SM register file is partitioned among all resident threads
  - The CUDA program can trade degree of thread block concurrency for amount of per-thread state
  - Registers, stack spill into “local” DRAM if necessary
- Each thread block has private access to a configurable amount of scratchpad memory
  - Pre-Fermi SM’s have 16 KB scratchpad only
  - The available scratchpad space is partitioned among resident thread blocks, providing another concurrency-state tradeoff



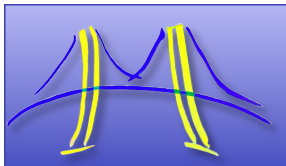
Memory, Memory, Memory

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



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

Image source: The CUDA programming guide



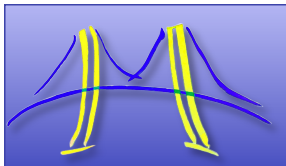
Thread-Block Synchronization

- Intra-block barrier instruction **`__syncthreads()`** for synchronizing accesses to **`__shared__`** memory
  - To guarantee correctness threads must **`__syncthreads()`** before reading values written by other threads
  - All threads in a block must execute the same **`__syncthreads()`** or the GPU will hang

"extern __shared__" allows the shared memory block to dynamically sized at run-time

```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

The function qualifier "__device__" indicates the function runs on the device, but unlike a kernel a "__device__" function cannot be called by a host … its' called by a kernel running on a device.



Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most CUDA programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:

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

- Software-managed scratchpad memory is allocated statically:

```
__shared__ int scratch[128];
scratch[threadIdx.x] = ... ;
```

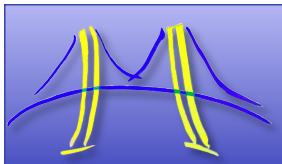
- ... or dynamically:

```
extern __shared__ int scratch[];

kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
```

The third argument is optional and gives the size in bytes of per-block shared memory

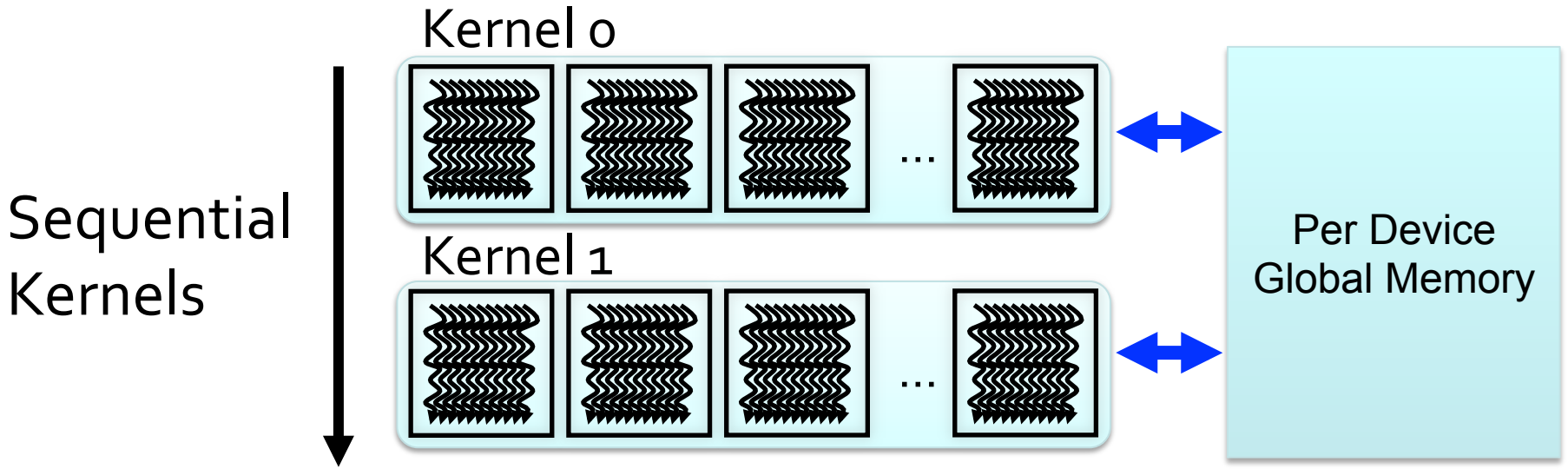
- Most intra-block communication is via shared scratchpad:

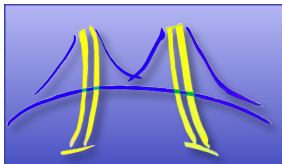
```
scratch[threadIdx.x] = ...;
__syncthreads();
int left = scratch[threadIdx.x - 1];
```



CUDA Memory Hierarchy

- Thread blocks in all Grids share access to a large pool of "Global" memory, separate from the Host CPU's memory.
  - Global memory holds the application's persistent state, while the thread-local and block-local memories are ephemeral
  - Global memory is much more expensive than local memories: O(100)x latency, O(1/50)x (aggregate) bandwidth
  - Registers and Cache multiply bandwidth, massive multithreading hides latency





CUDA memory hierarchy

- There are other read-only components of the Memory Hierarchy that exist due to the graphics heritage of CUDA
- The 64 KB CUDA **Constant Memory** resides in the same address space DRAM as global memory, but is accessed via special read-only 8 KB per-SM caches
- The CUDA **Texture Memory** also resides in DRAM's address space and is accessed via small per-SM read-only caches, but also includes interpolation hardware
  - This hardware is crucial for graphics performance, but only occasionally is useful for general-purpose workloads
- The behaviors of these caches are highly optimized for their roles in graphics workloads.

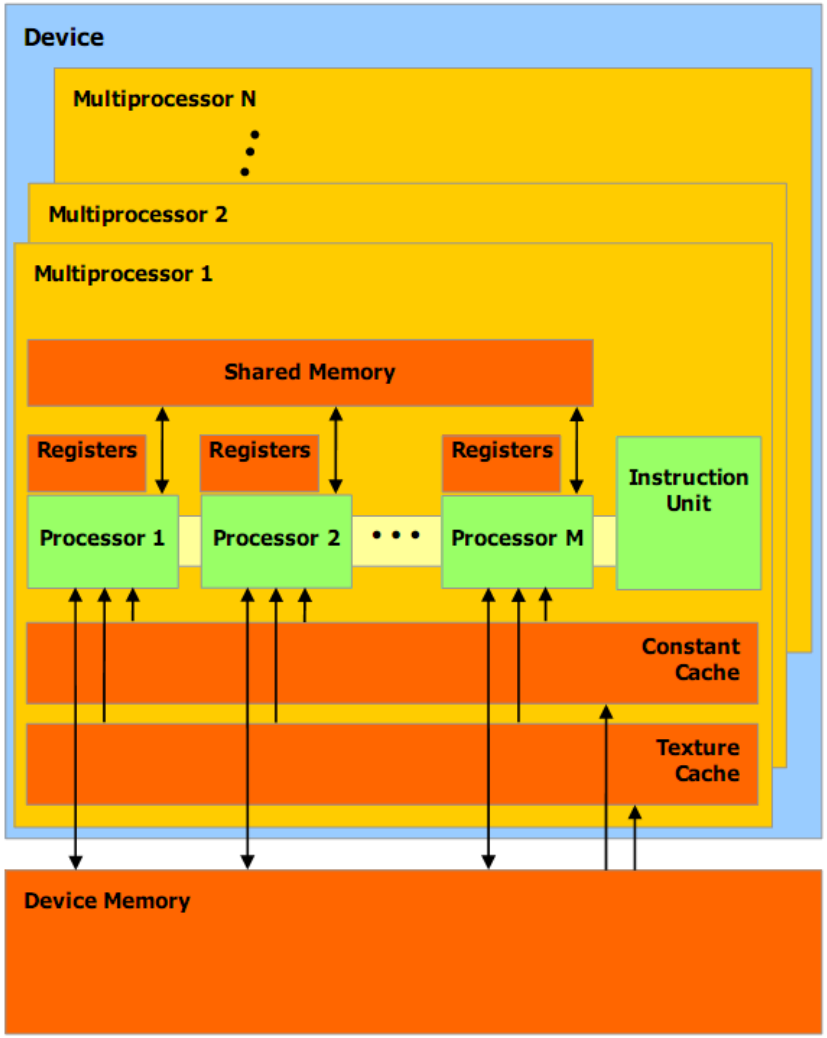
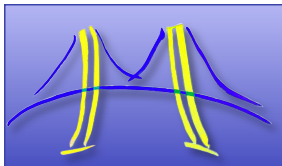
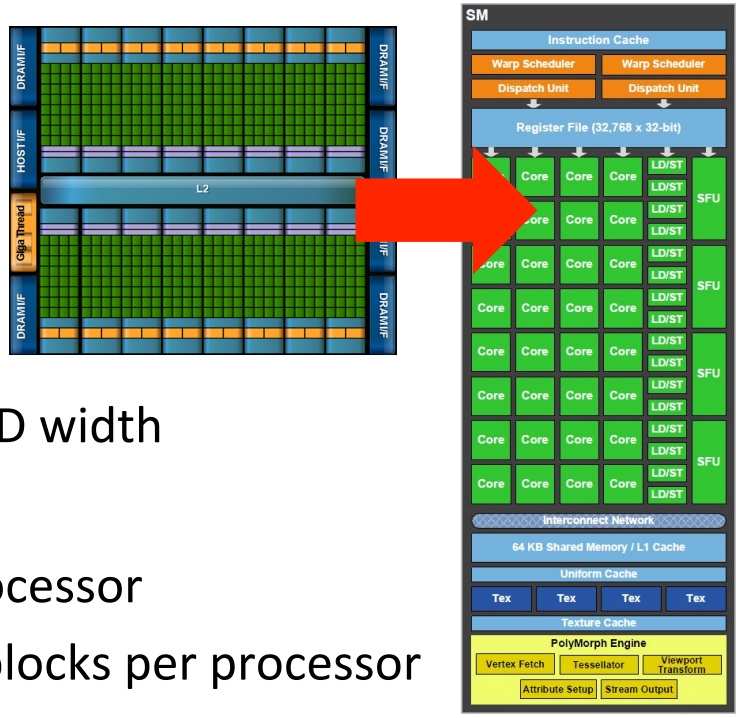


Image source: The CUDA programming guide



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 a processor
  - Peak efficiency requires multiple thread blocks per processor



CUDA Exercise 2

- Goal
  - Work with the CUDA memory hierarchy to optimize matrix multiplication.
- Problem
  - Start with the matrix multiplication program we provide to compute C= A*B in parallel
  - Parallelize with CUDA using the dot product for each element of C(I,j) as a CUDA-thread
  - Optimize performance by (1) putting rows of the A matrix in thread-local memory and (2) putting rows of the B matrix in thread-block shared memory.

Kernel example: void __global__ vfunc(const float *a, float *c, const int N);

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
cudaSetDevice(0);
cudaMalloc (&a,  sizeof(float) * LEN);
cudaMemcpy(a,&a[0],sizeof(float)*LEN,cudaMemcpyHostToDevice);
cudaMemcpy(&c[0],c, sizeof(float)*LEN,cudaMemcpyDeviceToHost);
cudaFree(a);
syncthreads();
extern __shared__ int scratch[];
```