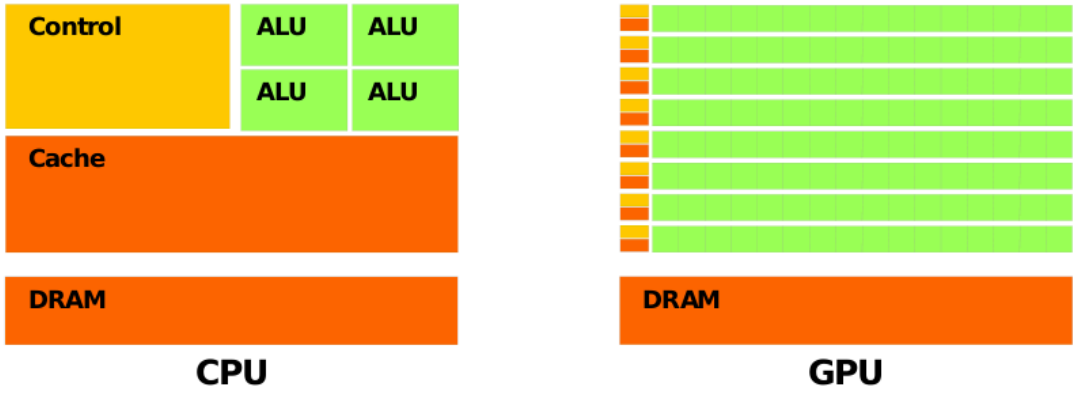
CUDA BASICS

CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions in CUDA

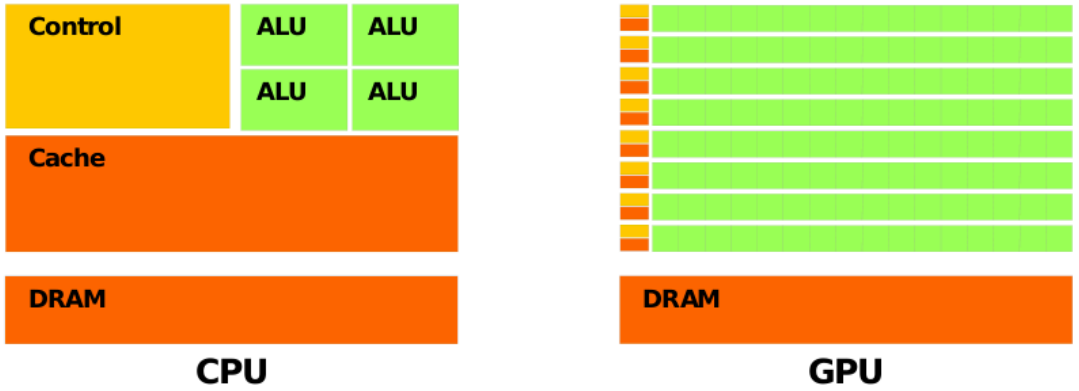
CPU vs. GPU architecture

- **CPUs are designed for general-purpose computing**
  - Sophisticated control to exploit ILP. ALU to exploit DLP.
  - Multicores (independent control flow) for TLP
  - Large caches to reduce impact of long latency memory accesses and to enable sharing in multicores



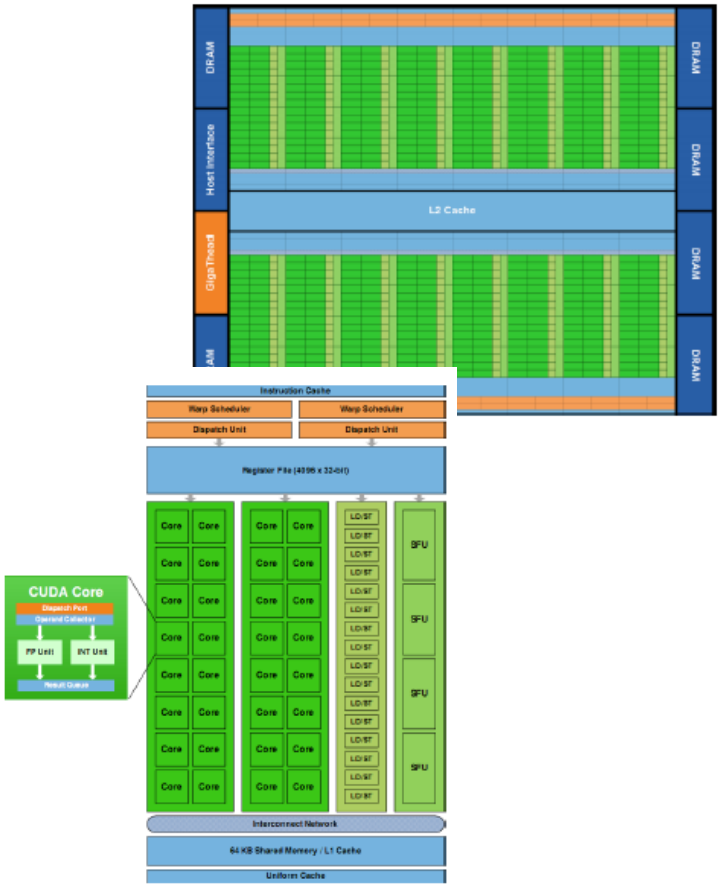
CPU vs. GPU architecture

- **GPUs are specialized for highly parallel, compute-intensive computation**
  - Simple control: same computation on all compute units
  - Massive number of threads to reduce impact of long latency memory accesses



NVIDIA GPU architecture: Parameters, terminology ....

- **NVIDIA Fermi**
  - 16 `streaming multiprocessors` (SM) interconnected via a 768k L2 cache crossbar
  - Each SM has 32 `cores` and 4096 `registers`, sharing `64 KB (48K L1 and 16K shared memory or viceversa)`
  - Each core has one `FP unit` and one `integer unit`
  - Each SM has 16 `load/store units`, allowing sixteen threads per clock to calculate memory addresses
  - Each SM has 4 `Special Function Units (SFU)` that execute complex instructions (sin, cosine, reciprocal, and square root)



NVIDIA GPU architectures

- **Exercise:**

  What are other GPU architectures ?

  Identify the architectural parameters for

  - NVIDIA Kepler
  - NVIDIA Pascal
  - NVIDIA Volta

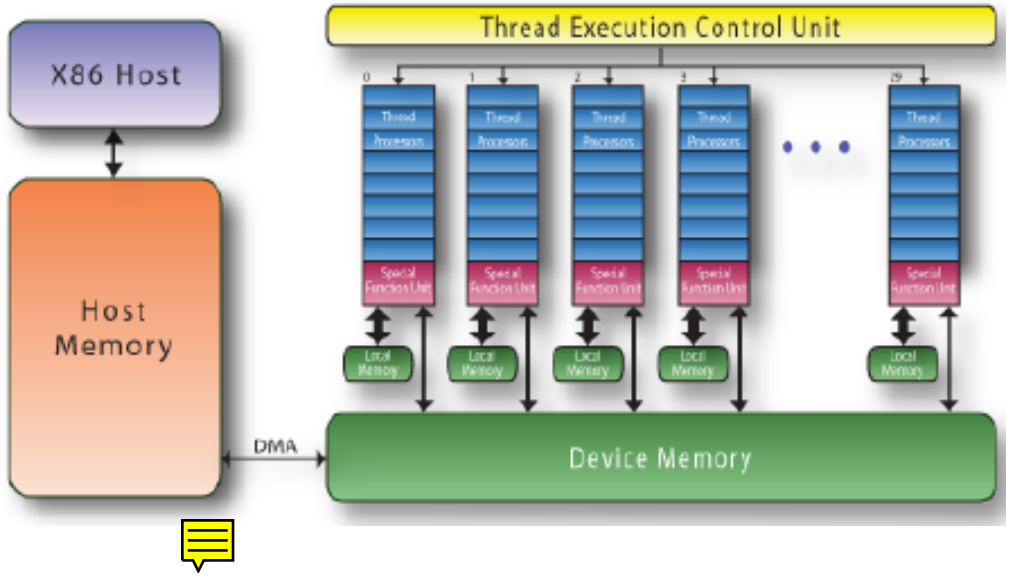
- **Exercise:**

  What is the GPU architecture for the LAB machines ?

  What are its main architectural parameters ?

CPU-GPU block diagram

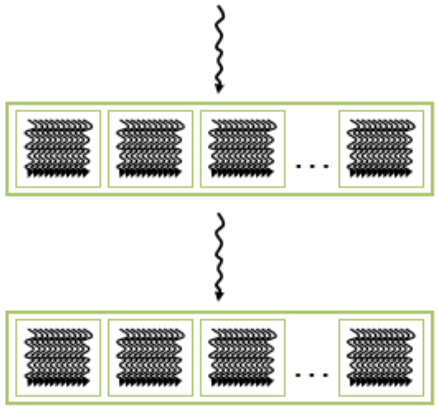
- **Separate address spaces between CPU (host memory) and GPU (device memory). Communication using IO commands and DMA memory transfers (e.g. PCI Express)**

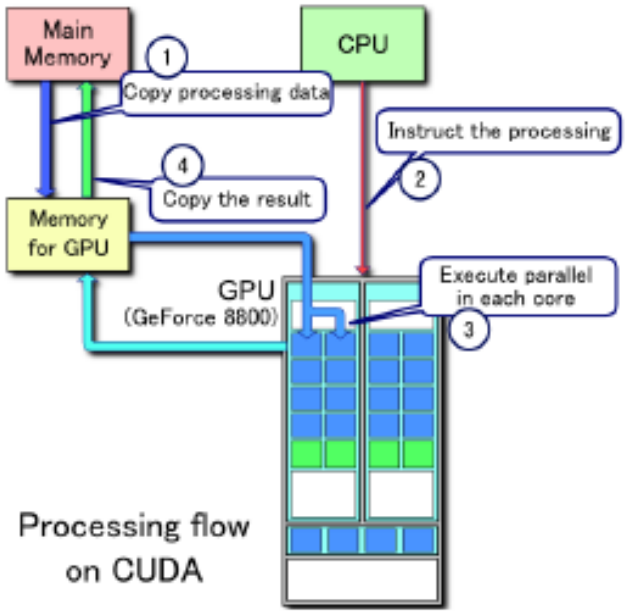


CPU-GPU execution flow

- **CPUs for sequential or modestly parallel parts, GPUs for highly parallel parts (kernels in CUDA)**

```
1) Copy processing data
2) Instruct the processing
3) Execute parallel computation
4) Copy back the result
```





CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions in CUDA

Device management

- **Application can query and select GPUs**

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
cudaGetDevice(int *device)
```

- **Device sharing**
  - Multiple threads can share a device
  - A single thread can manage multiple devices

Offloading kernel execution

- **Kernel**
  - Function that runs on the device and is called from host/device
- **Definition**
  - Using keyword `__global__`

```
__global__ void helloworld(void) {


}
```

Offloading kernel execution (cont.)

- **Kernel offloading for execution on GPU device**

```
...
helloworld<<<1,1>>>();
...
```

- **Triple angle brackets mark a "kernel launch" (call from host code to device code)**
  - We will cover the parameters inside the brackets in a few slides
  - The kernel launch is asynchronous (i.e. the host does not wait for the termination of the kernel execution, control returns to the CPU immediately)

Offloading kernel execution (cont.)

- **Host may need to wait before continuing execution**
  - The execution of `cudaDeviceSynchronize()` blocks the CPU until all preceding CUDA kernels have completed

```
__global__ void helloworld(void) {

}


int main(void) {

  helloworld<<<1,1>>>();
  cudaThreadSynchronize();

  printf("Hello World!\n");
  return 0;
}
```

Offloading to multi-GPU nodes

- **Any host thread can access all GPUs in the node (if more than one available) using the `cudaSetDevice` call**

```
__global__ void helloworld(void) {

}
int main(void) {
  int numDevs;
  cudaGetDeviceCount(&numDevs);
  for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    helloWorld<<<1,1>>>();
  }
  for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    cudaThreadSynchronize();
  }
  return 0;
}
```

CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions in CUDA

Parallel kernel

- **How to make use of the multiple streaming multiprocessors (SM) and threads inside each SM?**
  - Kernel replication using the arguments in the kernel launch
    - ✓ Replication in different SMs

```
helloWorld<<<N,1>>>();
```

    - ✓ Replication in different threads but in a single SM

```
helloWorld<<<1,N>>>();
```

Parallel kernel (II)

- **Each kernel instance can be uniquely identified, which can be used to make kernel instances cooperate in the resolution of a common problem**

```
__global__ void helloWorld(char* str) {
  int idx = blockIdx.x;
  str[idx] += idx;
}

int main (int argc, char *argv[]) {
  ...
  char str[] = "Hello World!";
  for(i = 0; i < 12; i++)
    str[i] -= i;
  ...
  helloWorld<<<12, 1>>>(d_str);
  ...
}
```

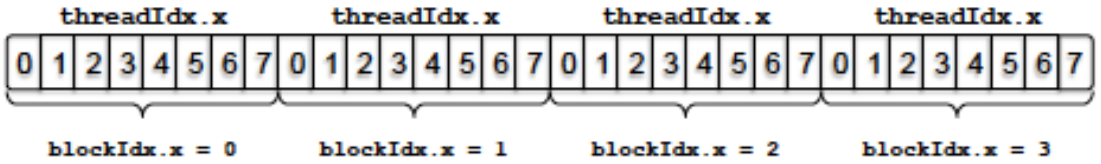
Parallel kernel (III)

- **Similarly if replication occurs at the threads level**

```
__global__ void helloWorld(char* str) {
  int idx = threadIdx.x;
  str[idx] += idx;
}
int main (int argc, char *argv[]) {
  ...
  char str[] = "Hello World!";
  for(i = 0; i < 12; i++)
    str[i] -= i;
  ...
  helloWorld<<<1, 12>>>(d_str);
  ...
}
```

Thread and Data Association

- **Indexing a vector with blocks and threads**
  - Consider indexing an array with one element per thread and M threads per block (e.g. 8)

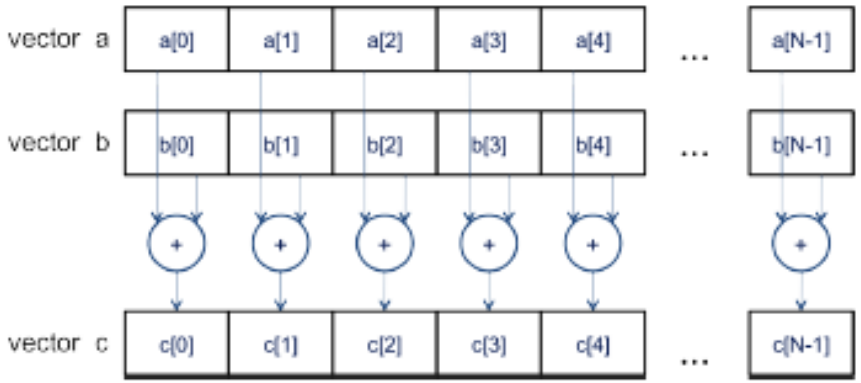


  - Then a unique index for each thread is given by

```
int index = threadIdx.x + blockIdx.x * M;
```

Example: vector addition (sequential code)

```
// Compute vector sum C = A+B
void vecAdd(float* a, float* b, float* c, int n) f
  for (i = 0, i < n, i++)
    c[i] = a[i] + b[i];
}

int main() f
  float a[N], b[N], c[N];

  // initialize a and b

  vecAdd(a, b, c, N);

  // display the results
  return 0;
}
```



Example: vector addition (kernel denition and invocation)

- **Kernel code on device**

```
// Each kernel invocation performs one pair-wise addition
__global__ void vecAddKernel(float* a_d, float* b_d, float* c_d, int n){

  int i = threadIdx.x + blockDim.x * blockIdx.x;
  c_d[i] = a_d[i] + b_d[i];

}
```

- **Kernel invocation on host**

```
__host__ int vectAdd(float* a, float* b, float* c, int n) {
  ...
  // Run ceil(n/256) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256), 256>>>(a, b, c, n);
  ...
}
```

Function declarations in CUDA

- **Functions in CUDA programs can be:**

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

AND DEVICE !!!!

  - A kernel function must return `void`

Example: vector addition (kernel code revisited)

- **What if `blockDim.x` does not divide `N`?**

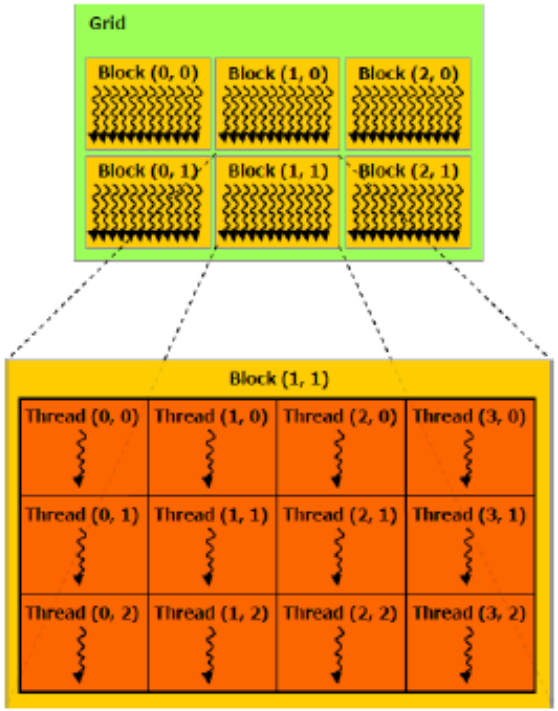
```
// Each thread performs one pair-wise addition
__global__ void vecAddKernel(float* a_d, float* b_d, float* c_d, int n) {

  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i<n)
    c_d[i] = a_d[i] + b_d[i];
}
```

- **Guarded execution to ensure that we are not going out from the `N` elements**

Identifying threads inside a grid

- In general, the GPU offers a grid of threads, divided into blocks (thread blocks), and each block is further divided into threads

- The grid of thread blocks can actually be partitioned into 1, 2 or 3 dimensions



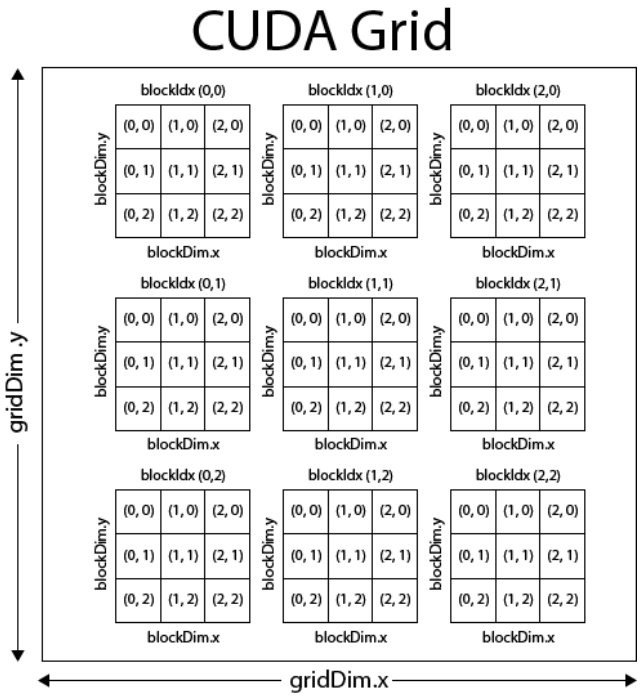
Identifying threads inside a grid

- **Each thread uses indices (1D, 2D or 3D) to decide what to do and data to work on (data decomposition)**

  `blockIdx`

  `threadIdx`

- **defined inside:**
  - a grid with `gridDim` blocks,
  - each block with `blockDim` threads



dim3 datatype

- **Data type to dene variables to hold block and grid dimensionalities**

```
__host__ int vectAdd(float* a, float* b, float* c, int n) {
  ...
  // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(ceil(n/256), 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernnel<<<DimGrid,DimBlock>>>(a, b, c, n);
  ...
}
```

Thread scheduling and execution

- **Threads in a block are divided in `32-thread warps` (scheduling units in SM)**
  - Example:

    if 3 blocks assigned to an SM, each with 256 threads, how many `warps` are there in an SM?

    – Each block is divided into 256/32 = 8 warps

    – There are 8 x 3 = 24 warps

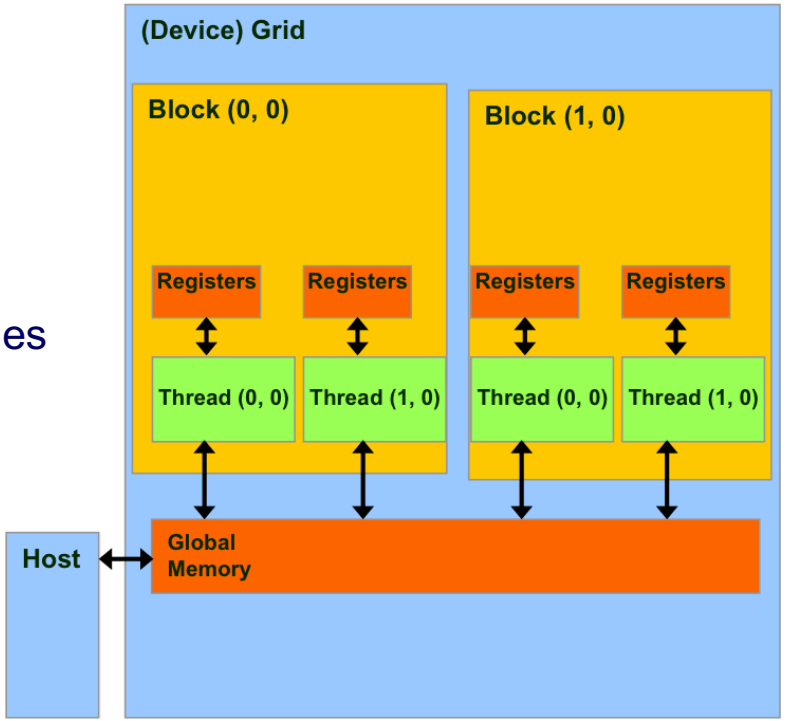
    ✓ At any point in time, only one of the 24 warps will be selected for instruction fetch and execution (multithreading)

CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions. in CUDA

Partial view of the device memory

- **Device code can**
  - read and write thread registers
  - read and write grid global memory
- **Host code can**
  - Allocate data in grid global memory
  - Transfer data between host and grid global memories



Basic device memory management

- **Global memory**
  - Contents visible to all threads
  - Variables in grid global memory declared using device attribute
  - Also dynamically allocatable using `cudaMalloc` and `cudaFree`

```
#define N 1000

__device__ int A[N]; // declared outside function and kernel bodies.

// Lifetime: application
__global__ kernel() {
  int tid = blockId.x * blockDim.x + threadId.x;
  A[tid]++;
}
```

Basic device memory management

- `cudaMalloc`
  - Allocates object in the device global memory
  - Two parameters: pointer to the allocated object, size of allocated object in bytes
- `cudaFree`
  - Frees object from device global memory
  - Parameter: pointer to object

Example: vector addition (host code)

```
int main() {
  float a[N], b[N], c[N]; // vectors in host memory
  float *dev_a, *dev_b, *dev_c; // pointers to vectors dynamically allocated in global memory

  // Allocate a, b and c on the device
  cudaMalloc( &dev_a, N * sizeof(float) ) ;
  cudaMalloc( &dev_b, N * sizeof(float) ) ;
  cudaMalloc( &dev_c, N * sizeof(float) ) ;
  ...
  // kernel invocation
  dim3 DimGrid(ceil(N/256), 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b, dev_c, N);
  ...
  // Free the memory allocated on device
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_c );
  return 0;
}
```

Basic device memory management (cont.)

- **Memory data transfer**

`cudaMemcpy(...)`

- ✓ Synchronous
  - – Blocks the CPU until the copy is complete
  - – Copy begins when all preceding CUDA calls have completed
- ✓ 4 parameters: pointer to destination, pointer to source, number of bytes to be copied and type of transfer

`cudaMemcpyAsync(...)`

- ✓ Asynchronous, does not block the CPU, allowing the overlap of data transfer and computation
- ✓ Additional stream parameter that if not 0 can be used to synchronize

Example: vector addition (host code) (cont'd)

```
int main() {
  float a[N], b[N], c[N]; // vectors in host memory
  float *dev_a, *dev_b, *dev_c; // pointers to vectors dynamically allocated in global memory

  cudaMalloc( &dev_a, N * sizeof(float) );
  cudaMalloc( &dev_b, N * sizeof(float) );
  cudaMalloc( &dev_c, N * sizeof(float) );
```

ALLOC

```
  // Initialize a and b in host memory (same as sequential)
  // Copy a and b from host to device memory
  cudaMemcpy( dev_a, a, N * sizeof(float), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, b, N * sizeof(float), cudaMemcpyHostToDevice );
```

COPY

```
  dim3 DimGrid(ceil(N/256), 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b, dev_c, N);
```

COMPUTE

```
  // Copy back c from device to host
  cudaMemcpy( c, dev_c, N * sizeof(float), cudaMemcpyDeviceToHost );
```

COPY

```
  // Display the results

  cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c );
  return 0;
}
```

FREE

Sharing data between GPUs

- **Options:**
  - Explicit copies via host
  - Peer-to-peer memory access:
    - ✓ Direct copy from pointer on device to pointer on another device

```
cudaMemcpyPeer( void *dst, int dstDevice, void *src, int
                srcDevice, size t count )
```

  - Unified Memory :
    - ✓ shared address space between devices and host (not covered in this course)

CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions in CUDA

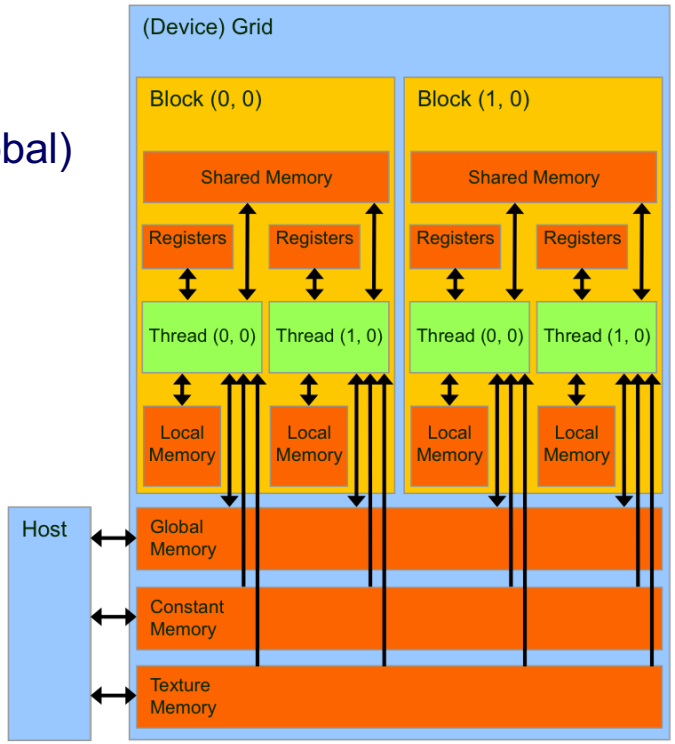
Synchronization: Threads vs. blocks

- **In CUDA it is not possible to synchronize threads belonging to different blocks**

- **Unlike blocks, threads within a block have mechanisms to:**
  - Synchronize: the instruction `__syncthreads()` forces all warps (i.e. all threads in a block) to wait until the rest have reached the same point
  - Communicate via memory:
    - ✓ let's take a look at a more complete view of the device memory ...

A more complete view of the device memory

- **Each thread can:**
  - R/W per-thread registers and local memory
  - R/W per-block shared memory (100 times faster than global)
  - R/W per-grid global memory
  - Read only per-grid constant and texture memories
- **Host code can:**
  - R/W global, constant, and texture memories



A more complete view of the device memory

- **Shared memory**
  - Within a block, used to share data among threads
  - Allocated per block, data is not visible to threads in other blocks
- **Declared using** `__shared__`

```
#define N 1000

__global__ kernel() {
  __shared__ int A[N]; // declared inside kernel bodies.

  // Scope: block. Lifetime: kernel call
  int tid = threadId.x;
  A[tid]++;
  ...
}
```

CUDA Basics

- **Outline**
  - GPU architecture
  - Devices, kernel definition and offloading
  - Blocks, threads and indexing
  - Accessing (global) memory
  - Cooperating threads and shared memory
  - Reductions in CUDA

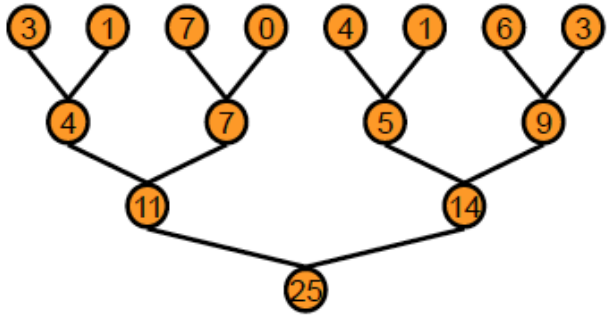
Parallel Reduction

- **Common and important data parallel primitive**
  - Easy to implement in CUDA
    - ✓ Harder to get it right

- **Serves as a great optimization example**
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

Parallel Reduction

- **Tree-based approach used within each thread block**



- **Need to be able to use multiple thread blocks**
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- **But how do we communicate partial results between thread blocks?**

Problem: Global Synchronization

- **If we could synchronize across all thread blocks, could easily reduce very large arrays, right?**
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- **But CUDA has no global synchronization. Why?**
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- **Solution: decompose into multiple kernels**
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Reduction #1: Interleaved Addressing (I)

```
__global__ void reduce0(int* g_idata, int* g_odata){
  extern __shared__ int sdata[];
  // each thread loadsone element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if(tid % (2*s) == 0){
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Copy to shared memory

Reduction in shared memory

Thread synchronization

Copy back to global memory

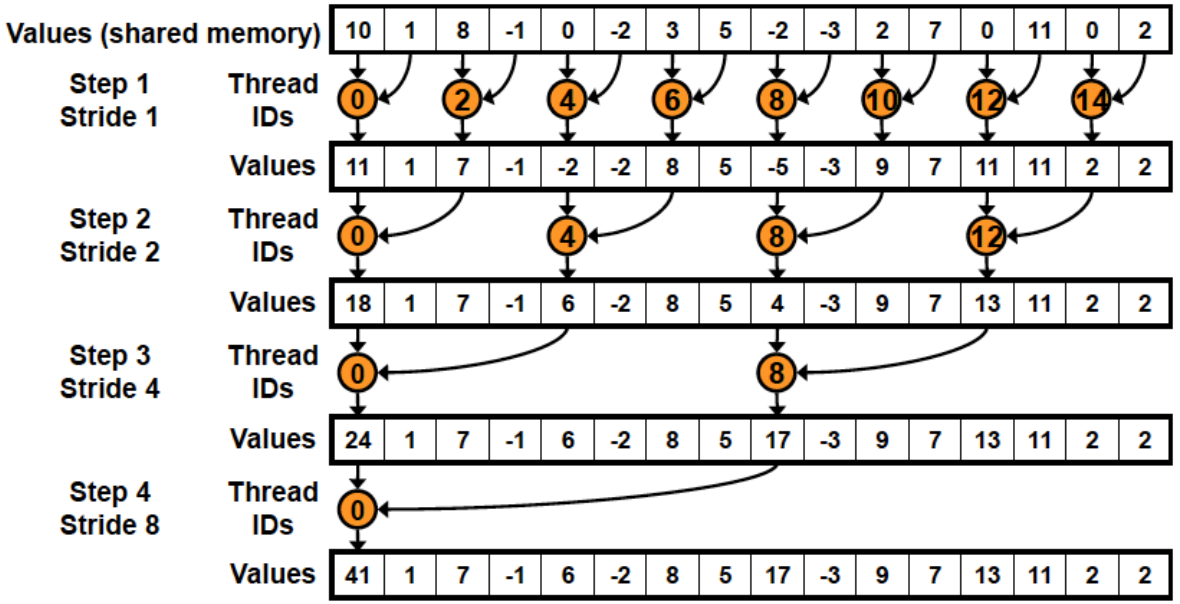
Reduction #1: Interleaved Addressing (II)

```
// PRE:
// dA is an array allocated on the GPU
// N <= len(dA) is a power of two (N >= BLOCKSIZE)
// POST: the sum of the first N elements of dA is returned
template<size_t blockSize, typename T>
T CPUReduction(T* dA, size_t N)
{
    T tot = 0.;
    size_t n = N;
    size_t blocksPerGrid = std::ceil((1.*n) / blockSize);

    T* g_out1, g_out2;
    cudaMalloc(&g_out1, sizeof(T) * blocksPerGrid);
    cudaMalloc(&g_out2, sizeof(T) * blocksPerGrid);

    T* from = dA;
    T* to = g_out1;

    // Code continues on next slide !!!
```

Reduction #1: Interleaved Addressing (III)

```
    do
    {
        blocksPerGrid   = std::ceil((1.*n) / blockSize);
        reduce0<blockSize><<<blocksPerGrid, blockSize>>>(from, to);
        tmp=from; from = to;
        to = tmp;
        n = blocksPerGrid;
    } while (n > blockSize);

    if (n > 1)
        reduce0<blockSize><<<1, blockSize>>>(tmp, tmp);

    cudaDeviceSynchronize();

    cudaMemcpy(&tot, tmp, sizeof(T), cudaMemcpyDeviceToHost);

    cudaFree(tmp);
    return tot;
}
```

Parallel Reduction: Interleaved Addressing



Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int* g_idata, int* g_odata){
  extern __shared__ int sdata[];
  // each thread loadsone element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for(unsigned ints=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0){
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

Reduction #2: Interleaved Addressing

- Just replace divergent branch in inner loop:

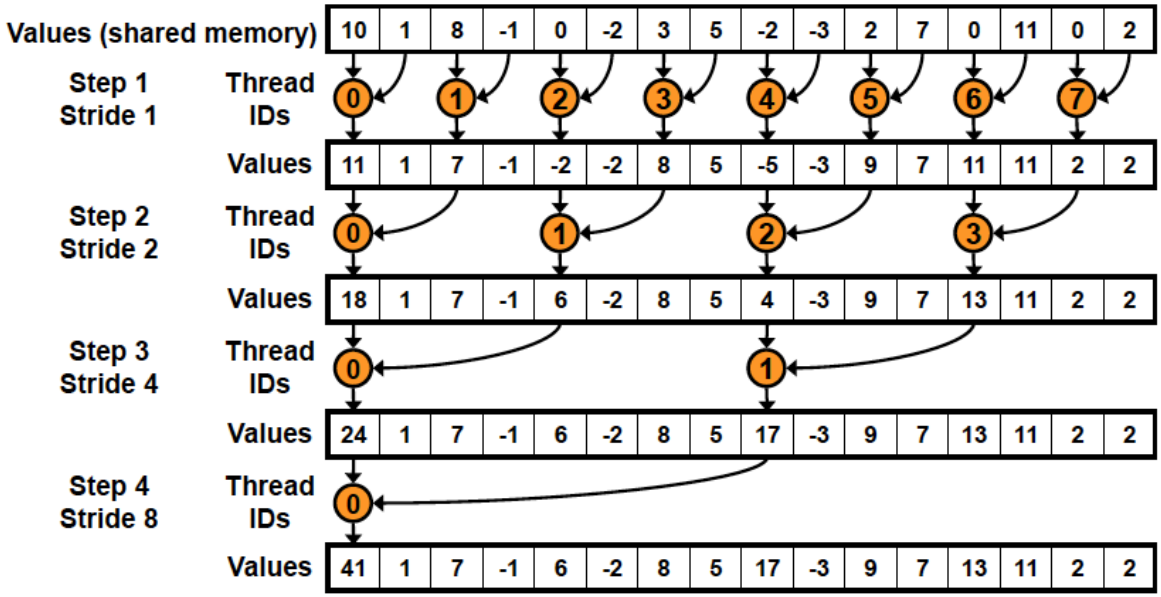
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

- With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

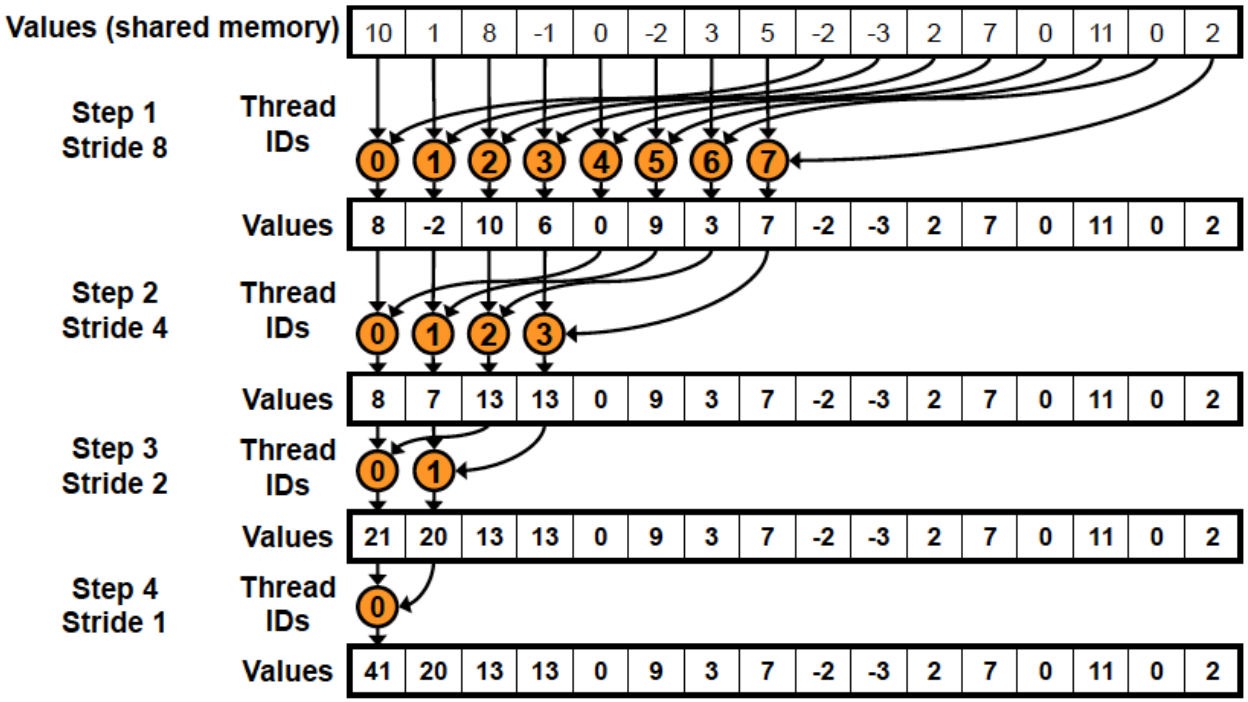
Parallel Reduction: Interleaved Addressing

- **New Problem: Shared Memory Bank Conflicts**



Parallel Reduction: Sequential Addressing

- **Sequential Addressing is conflict free**



Reduction #3: Sequential Addressing

- Just replace strided indexing in inner loop:

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

- With reversed loop and threadID-based indexing:

```
for(unsigned int s = blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

Idle Threads

- **Problem: Half of the threads are idle on first loop iteration!**
  - This is wasteful…

```
for(unsigned int s= blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

Reduction #4: First Add During Load

- **Halve the number of blocks, and replace single load:**

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i= blockIdx.x*blockDim.x+ threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

- **With two loads and first add of the reduction:**

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Unrolling the Last Warp

- **As reduction proceeds, # "active" threads decreases**
  - When s <= 32, we have only one warp left
- **Instructions are SIMD synchronous within a warp**
- **That means when s <= 32:**
  - We don't need to __syncthreads()
  - We don't need "if (tid < s)" because it doesn't save any work

- **Let's unroll the last 6 iterations of the inner loop**

Reduction #5: Unroll the Last Warp

- **Unroll latest 6 iterations**

```
__device__ void warpReduce(volatile int* sdata, int tid) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
```

Note: This saves useless work in *all* warps, not just the last one!

Without unrolling, all warps execute every iteration of the for loop and if statement

- **Then**

```
for(unsigned int s=blockDim.x/2; s>32; s>>=1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
if (tid < 32)warpReduce(sdata, tid);
```

Reduction #6: Template the unrolling with block size

- **Specify block size as a function template parameter:**

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

  - Function `reduce5` code:

```
if (blockSize >= 512)
  if (tid < 256) { sdata[tid] += sdata[tid + 256]; }__syncthreads(); }
if (blockSize >= 256)
  if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128)
  if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }
if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

- **Warp-level function:**

```
template <unsigned int blockSize>
__device__ void warpReduce(volatileint* sdata,int tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
  if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
  if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
  if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

Note: all code in RED will be evaluated at compile time. Results in a very efficient inner loop!

Invoking Template Kernels

- **Don't we still need block size at compile time?**
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
{
  case 512:
    reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 256:
    reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 128:
    reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 64:
    reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 32:
    reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 16:
    reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 8:
    reduce5< 8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 4:
    reduce5< 4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 2:
    reduce5< 2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 1:
    reduce5< 1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```

Reduction #7: Multiple Adds / Thread

- **Replace load and add of two elements:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

- **With a while loop to add as many as necessary:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
  sdata[tid] += g_idata[i] + g_idata[i+blockSize];
  i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing

Putting it all together

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
  if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
  if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
  if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;
  while (i < n){sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
  __syncthreads();
  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
  if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }
  if (tid < 32)warpReduce(sdata, tid);
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

CUDA BASICS