King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 13: GPU Compute APIs, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #7 + #8 (until Oct 23)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 7** (*Convolution*)
- Programming Massively Parallel Processors book (4th edition), **Chapter 8** (*Stencil*)

Read (optional):

- Inline PTX Assembly in CUDA: `Inline_PTX_Assembly.pdf`
- Dissecting GPU Architectures through Microbenchmarking:

Volta: `https://arxiv.org/abs/1804.06826`
Turing: `https://arxiv.org/abs/1903.07486`

`https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf`

Ampere: `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/`

Next Lectures

Lecture 14: Wed, Oct 12

no lectures on Oct 16 and Oct 19 ! (mid-semester break and IEEE VIS conference)

Lecture 15: Sun, Oct 23

Lecture 16: Wed, Oct 26

Lecture 17: Sun, Oct 30

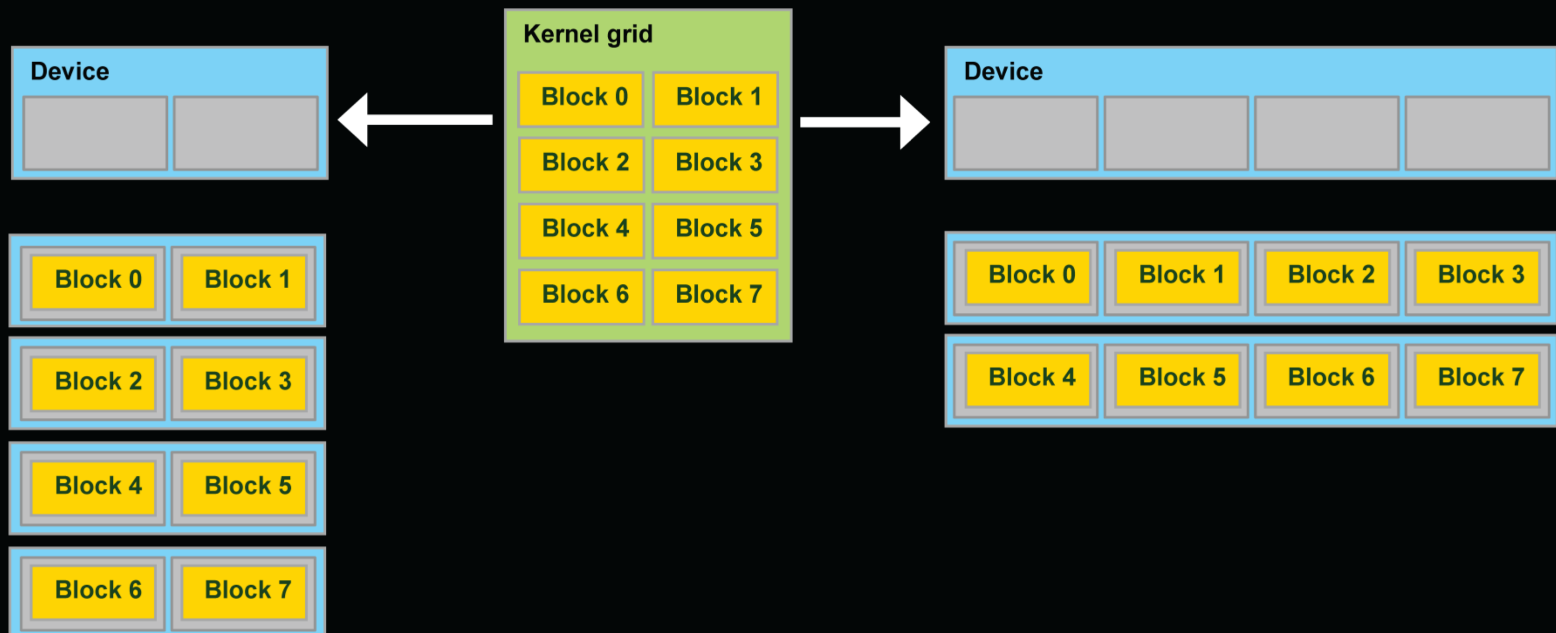
Lecture 18: Tue, Nov 1 (make-up lecture; 16:00 – 17:15 ?)

Lecture 19: Wed, Nov 2

GPU Compute APIs

Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors



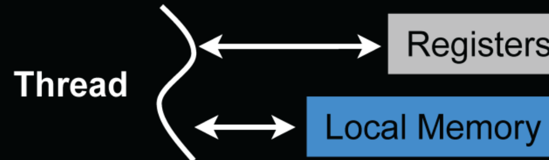
© 2008 NVIDIA Corporation.

NVIDIA

nVISION 08 THE WORLD OF VISUAL COMPUTING

Kernel Memory Access

- **Per-thread**

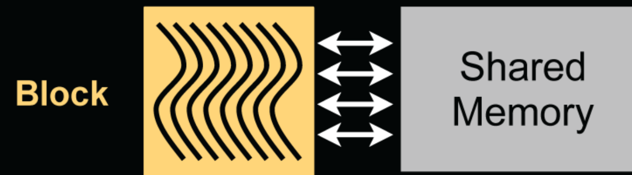


On-chip

Off-chip, ~~uncached~~

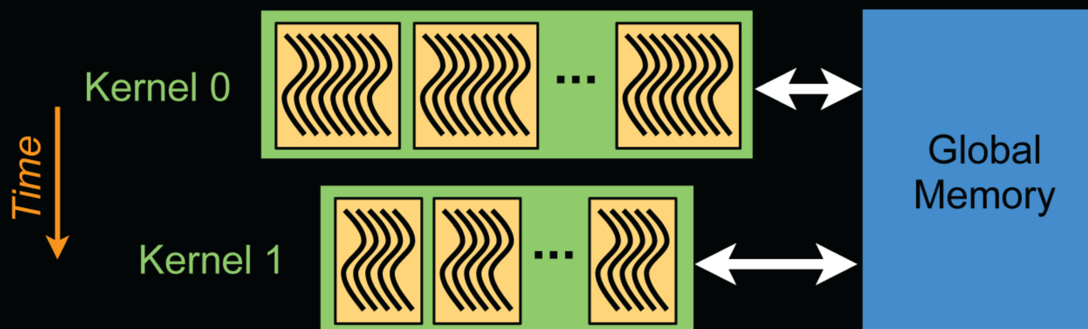
cached on Fermi or newer!

- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- ~~Uncached~~
- Persistent across kernel launches
- Kernel I/O

cached on Fermi or newer!

NVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA

Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	~~No~~* YES	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	~~No~~* YES	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

* cached on Fermi or newer!

© NVIDIA Corporation 2009

(Memory) State Spaces

PTX ISA 7.8 (Chapter 5)

Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-cluster[5]
`.tex`	No[4]	Yes, via driver	RO	Context

Notes:

[1] Variables in `.const` and `.global` state spaces are initialized to zero by default.

[2] Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

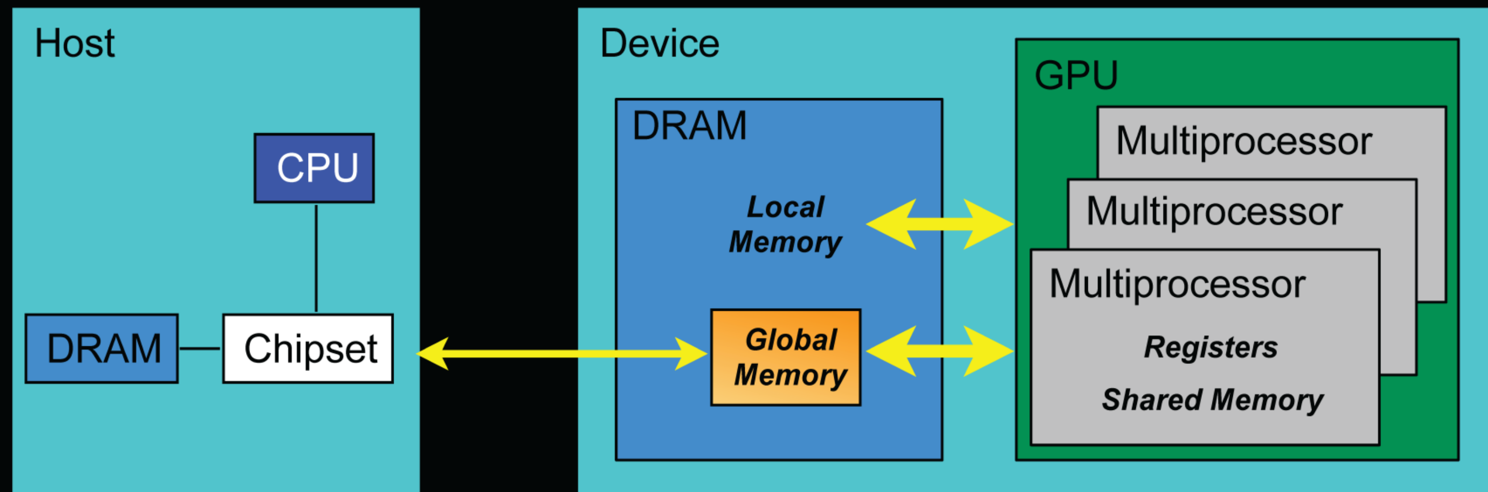
[4] Accessible only via the `tex` instruction.

[5] Visible to the owning CTA and other active CTAs in the cluster.

Managing Memory

Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- **CPU and GPU have separate memory spaces**
- **Host (CPU) code manages device (GPU) memory:**
  - **Allocate / free**
  - **Copy data to and from device**
  - **Applies to *global* device memory (DRAM)**



© 2008 NVIDIA Corporation.
NVIDIA

GPU Memory Allocation / Release

- cudaMalloc(void ** pointer, size_t nbytes)
- cudaMemset(void * pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d,  nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

nvision 08
THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA.

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

© 2008 NVIDIA Corporation.

Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```

© 2008 NVIDIA Corporation.

Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```

Host

a_h

b_h

nVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA.

Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

Host
a_h
b_h

Device
a_d
b_d

NVISION 08
THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA.

Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```

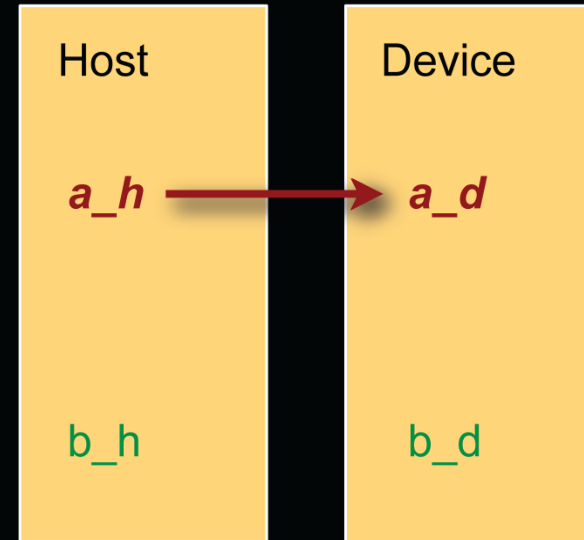

NVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA

Data Movement Example

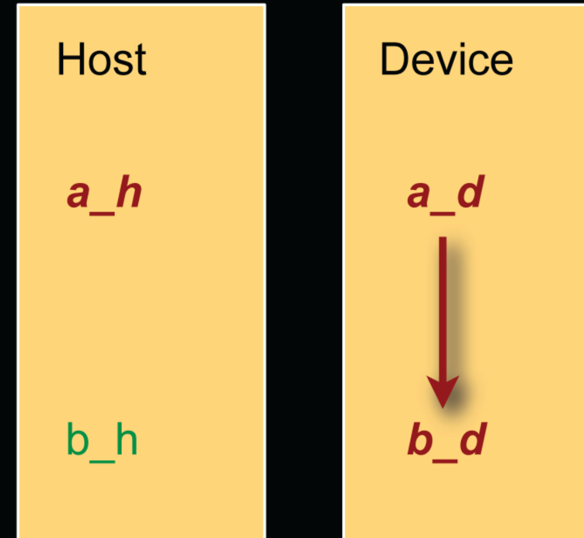
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



© 2008 NVIDIA Corporation.

Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



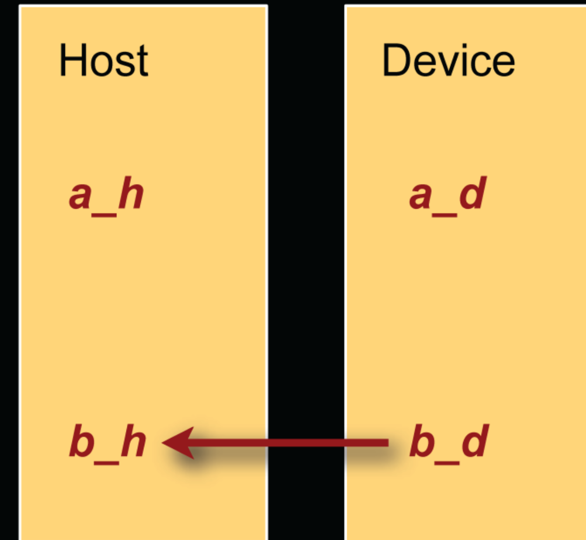
NVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA

Data Movement Example

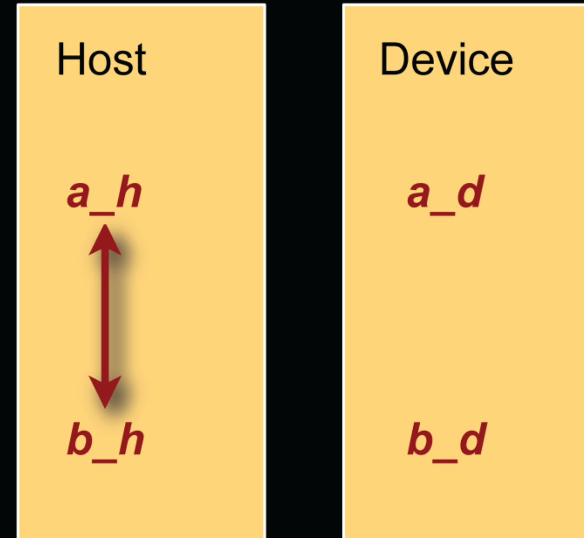
```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



© 2008 NVIDIA Corporation.
NVIDIA
NVISION 08
THE WORLD OF VISUAL COMPUTING

Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



NVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA

Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

Host

Device

NVISION 08
THE WORLD OF VISUAL COMPUTING
© 2008 NVIDIA Corporation.
NVIDIA.

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory — except: (*) and (**)
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - (Not recursive) — recursion supported on __device__ functions from cc. 2.x (i.e., basically on *all* current GPUs)
  - No static variables
- Function arguments automatically copied from host to device

(*) “unified memory programming” introduced with CUDA 6 (cc. 3.x +): allocate memory with cudaMallocManaged(); uses automatic migration

(**) also: mapped pinned (page-locked) memory (“zero-copy memory”) : allocate memory with cudaMallocHost(); beware of low performance!!

Function Qualifiers

- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void
- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code
  - __host__
    - Function called from host and executed on host (default)
    - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code

© 2008 NVIDIA Corporation.

Variable Qualifiers (GPU code)

- __device__
  - Stored in global memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - What doesn't fit in registers spills to "local" memory

CUDA 6+: __managed__ (with __device__) for managed memory (unified memory programming)

NVISION 08 THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA.

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

NVISION 08
THE WORLD OF VISUAL COMPUTING

© 2008 NVIDIA Corporation.

NVIDIA.

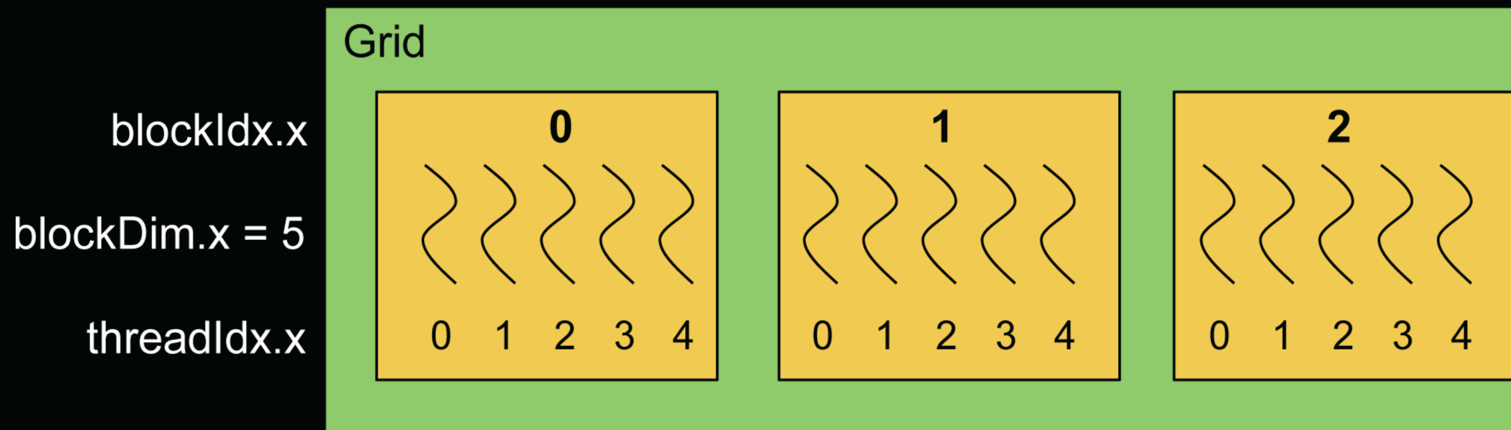
CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

© 2008 NVIDIA Corporation.

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

blockIdx.x*blockDim.x + threadIdx.x

© 2008 NVIDIA Corporation.

NVIDIA

NVISION 08 THE WORLD OF VISUAL COMPUTING

Increment Array Example

CPU program

```
void inc_cpu(int *a, int N)
{
  int idx;

  for (idx = 0; idx<N; idx++)
     a[idx] = a[idx] + 1;
}

int main()
{
   ...
   inc_cpu(a, N);
}
```

CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
  int idx = blockIdx.x * blockDim.x
               + threadIdx.x;
  if (idx < N)
     a[idx] = a[idx] + 1;
}

int main()
{
  ...
  dim3 dimBlock (blocksize);
  dim3 dimGrid( ceil( N / (float)blocksize)  );
  inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

© 2008 NVIDIA Corporation.

Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA
- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

© 2008 NVIDIA Corporation.

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- ~~cudaThreadSynchronize()~~
  - blocks until all previous CUDA calls complete

CUDA 4.x or newer:
cudaDeviceSynchronize() *and*
cudaStreamSynchronize()

© 2008 NVIDIA Corporation.

Host Synchronization Example

```
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
```

nVISION 08
THE WORLD OF VISUAL COMPUTING
© 2008 NVIDIA Corporation.
NVIDIA.

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

© NVIDIA Corporation 2006

Synchronization

- Threads in the same block can communicate using shared memory
- __syncthreads()
  - Barrier for threads only within the current block
- __threadfence()
  - Flushes global memory writes to make them visible to all threads

Plus newer sync functions, e.g., from compute capability 2.x on:
__syncthreads_count(), __syncthreads_and/or(),
__threadfence_block(), __threadfence_system(), …

Now: *Must* use versions with _sync suffix, because of
Independent Thread Scheduling (compute capability 7.x and newer)!

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

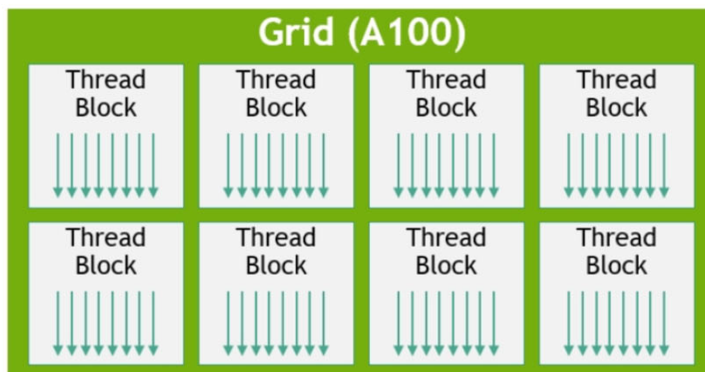
```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

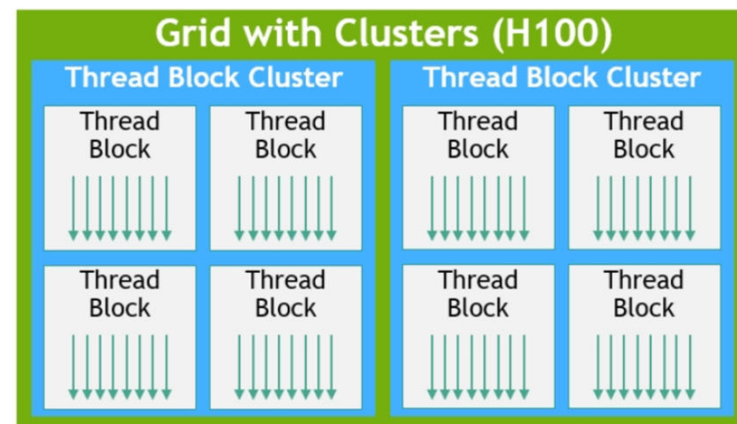
```
coalesced_group g = coalesced_threads();



int res;
if (g.thread_rank() == 0)
   res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
   res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

New in CC 9.0: Thread Block Clusters

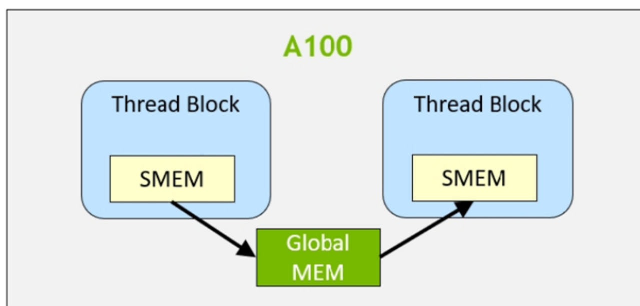
New thread hierarchy level!

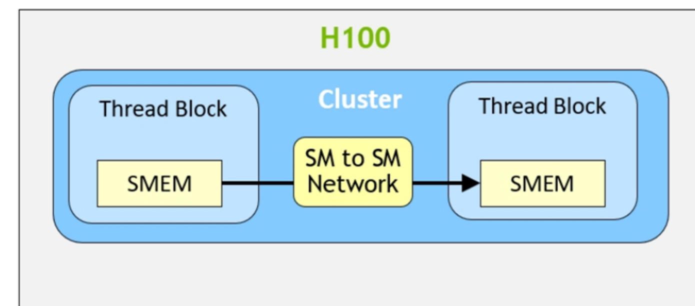




*all threads of a block are on the same **SM !***

*all blocks of a cluster are on the same **GPC !***



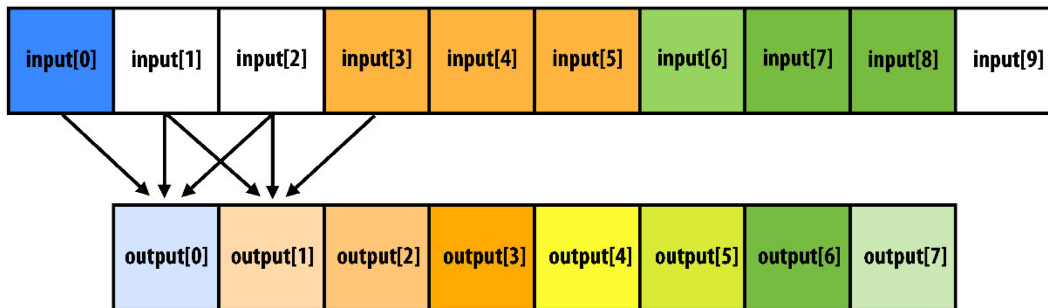


Code Examples

Example #1: 1D Convolution

Example #1: 1D Convolution

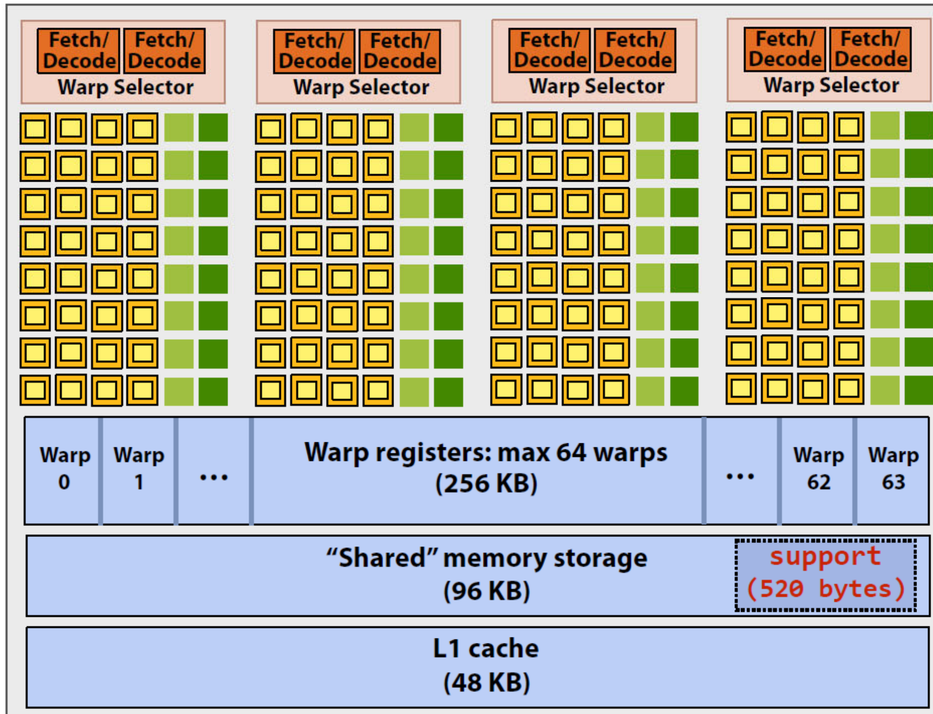
1D Convolution with 3-tap averaging kernel
(every thread is averaging three inputs)




```
output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Running on a GP104 (Pascal) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

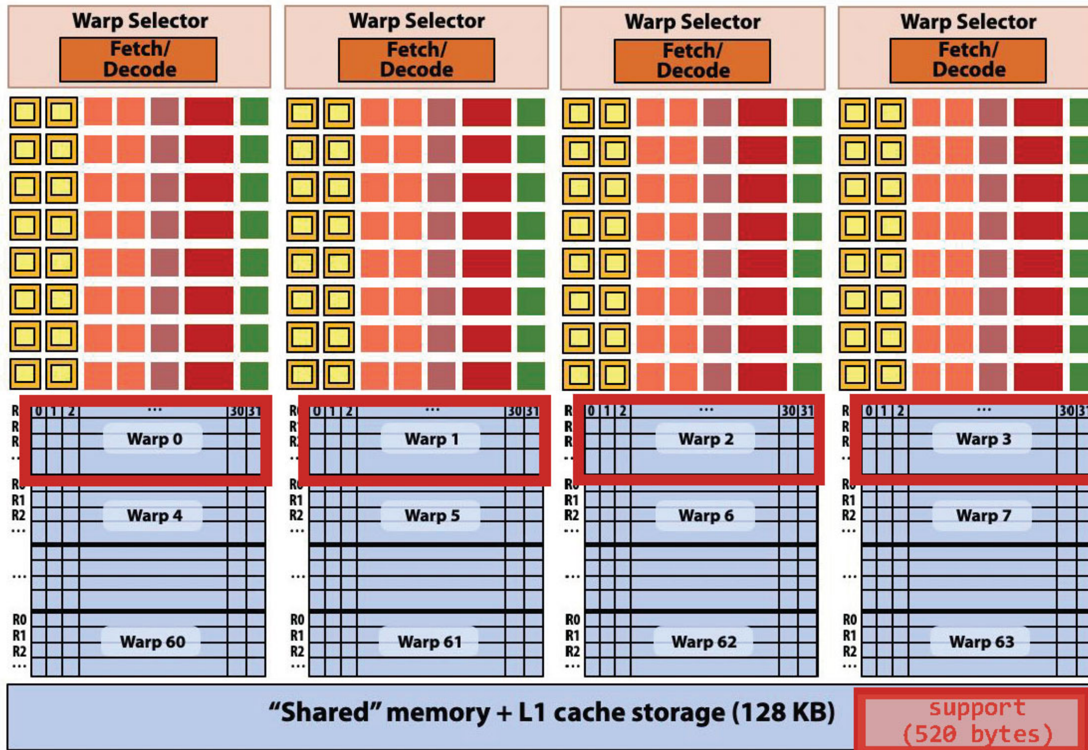
A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SM core operation each clock:

- **Select up to four runnable warps from 64 resident on SM core (thread-level parallelism)**
- **Select up to two runnable instructions per warp (instruction-level parallelism)** * *(but no ALU dual-issue!)*

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

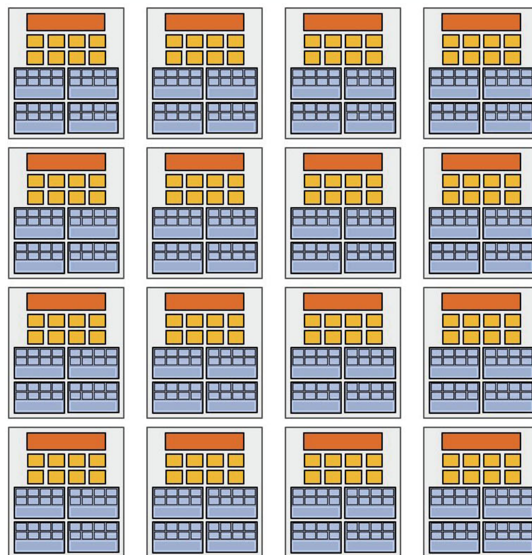
SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

Code on Same SM Arch. But Different #SMs

Assigning work



High-end GPU
(16 cores)

Mid-range GPU
(6 cores)

Desirable for CUDA program to run on all of these GPUs without modification

Note: there is no concept of `num_cores` in the CUDA programs I have shown you. (CUDA thread launch is similar in spirit to a forall loop in data parallel model examples)

(could now be up to 144 SMs, etc., …)

Thank you.