GPU Programming

Keshav Pingali

Some slides borrowed from David Kirk and Wen-Mei Hwu, and from Ruetsch and Oster

Terminology

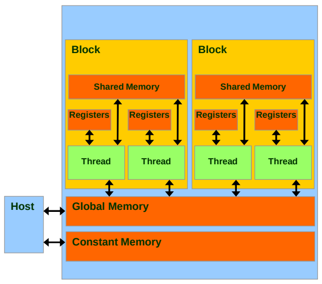
- Graphics Processing Unit (GPU)
  - special processors (accelerators) designed to speed up graphics applications
- General-purpose GPUs (GPGPU)
  - GPUs that have been massaged so that they can be used for both graphics and general-purpose applications
  - we will just refer to them as GPU's
- Compute Unified Device Architecture (CUDA)
  - NVIDIA programming model for their GPU's
- Open Computing Language (OpenCL)
  - emerging standard for programming heterogeneous processors: multicores + GPUs + other accelerators
- Kernel
  - a function/loop that is executed on GPU
  - a program will usually consist of a sequence of kernels interspersed with code that is executed on the host device (CPU)

Key features of GPUs

- Lots of threads
  - (eg) NVIDIA Fermi streaming processor has
    - 512 cores
    - 24,576 threads
  - lightweight threads: managed by hardware, start-up cost is small
- SIMT execution
  - groups of threads (*warp*) operate in SIMD
  - Siamese twins: 32 threads joined at hip
  - threads in warp are co-scheduled for execution
- Latency-tolerant architecture
  - processor time-slices between warps to mask memory and synchronization latencies
  - cf: time-sharing, dataflow

Exposed Memory Hierarchy

- **Global memory:**
  - Read/written by host
  - Read/written by all GPU threads
  - Used to transfer data back and forth between host and GPU
  - Relatively slow: 400-800 cycles
- **Constant memory:**
  - Read/written by host
  - Read by GPU threads
  - Used to transfer read-only information
- **Shared memory:**
  - Read/written by groups of threads called thread blocks or just blocks
  - Like a software managed L1 cache
  - Faster than global memory:1-4 cycles
- **Registers:**
  - Read/written by thread
  - Private to each thread



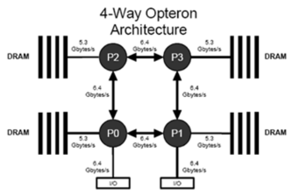
In principle, global memory + registers are enough.
Shared-memory: intermediate level of memory hierarchy

4

Note on hierarchical thread organization

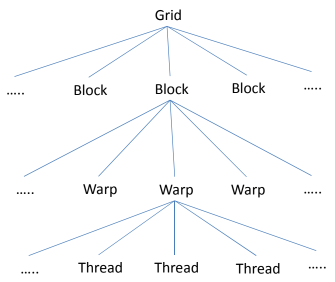
- Even on multicore processors
  - threads are organized physically in a hierarchy
  - storage is associated with multiple levels of this hierarchy
    - (eg) threads in same chip can share L2 or L3 cache
- Difference
  - data is automatically moved by hardware from one cache to another
  - so association between threads and cache does not have to be exposed to programming model
- Exposed memory hierarchy of GPU
  - data movement must be orchestrated by programmer
  - so association between threads and storage is exposed to programming model

CORE 1
CORE 2
CORE 3
CORE 4
L2
L3
MEMORY
I/O



Hierarchical Organization of Threads

Hierarchy reflects both SIMT and exposed memory hierarchy



Programming model implication

Grid has global memory.
Blocks in grid are usually independent.

Block has id, per-block shared memory.
Detail: block id can be 1D,2D,3D.
CUDA: blockIdx.x, blockIdx.y,blockId.z
CUDA: sync_threads for synchronizing all threads in block.

Not reflected in programming model.
Performance: thread/memory divergence

Thread has id, registers, PC,
thread-private local memory.
Detail: thread id can be 1D,2D,3D.
CUDA: threadIdx.x,threadIdx.y,threadIdx.z

6

First CUDA example

- Let us write a program to compute
  - C = A+B
  - where A,B,C are arrays
- Thread i will compute
  - C[i] = A[i]+B[i]

Basic CUDA program structure

```
int main (int argc, char **argv ) {

        1. Allocate memory space in device (GPU) for input and output data
        2. Create input data in host (CPU)

        3. Copy input data to GPU

iterate 4. Call "kernel" routine to execute on GPU
        (with CUDA syntax that defines no of threads and their logical organization)

        5. Transfer output data from GPU to CPU

        6. Free memory space in device (GPU)
        7. Free memory space in host (CPU)

        return value;
}
```

Kernel routine:
marching orders for one thread (cf. pThreads)
use threadId and blockId to give different work to different threads
Call to a kernel function is asynchronous from CUDA 1.0 on.
Synchronization for blocking host till all previous CUDA calls complete:
cudaThreadSynchronize()

8

CUDA Function Declarations

	Executed on the:	Only callable from the:
`_device_ float DeviceFunc()`	device	device
`_global_ void KernelFunc()`	device	host
`_host_ float HostFunc()`	host	host

- Executed on host, callable from device: not supported
- `_global_` defines a kernel function, must return `void`
- `_device_` and `_host_` can be used together

9

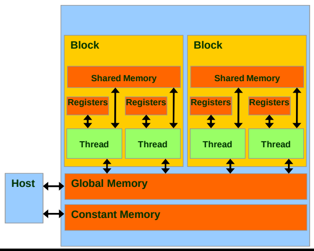
```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main (int argc, char **argv ) {

    int size = N *sizeof( int);
    int  *a, *b, *c, *devA, *devB, *devC;

    cudaMalloc( (void**)&devA, size) );
    cudaMalloc( (void**)&devB, size );
    cudaMalloc( (void**)&devC, size );

    a = (int*)malloc(size); b = (int*)malloc(size);c = (int*)malloc(size);

    cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( devB, b size, cudaMemcpyHostToDevice);

    vecAdd<<<1, N>>>(devA, devB, devC);

    cudaMemcpy( &c, devC size, cudaMemcpyDeviceToHost);
    cudaFree( dev_a);
    cudaFree( dev_b);
    cudaFree( dev_c);
    free( a ); free( b ); free( c );

    return (0);
}
```

10

1. Allocating memory in GPU global memory for data

Use CUDA malloc routines:

```
int size = N *sizeof( int); // space for N integers
int *devA, *devB, *devC; // devA, devB, devC ptrs

cudaMalloc((void**)&devA, size);
cudaMalloc((void**)&devB, size );
cudaMalloc((void**)&devC, size );
```



11

2. Creating input data in “host” (CPU)

Use regular C malloc routines or static variables and compute:

```
int *a, *b, *c;
…
a = (int*)malloc(size);
b = (int*)malloc(size);
c = (int*)malloc(size);
```

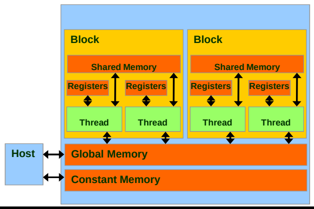
12

3. Transferring data from host (CPU) to device (GPU)

Use CUDA routine cudaMemcpy

cudaMemcpy(devA, &A, size, cudaMemcpyHostToDevice);
cudaMemcpy(devB, &B, size, cudaMemcpyHostToDevice);

where devA and devB are pointers to destination in device and A and B are pointers to host data



13

4. Calling "kernel" routine to execute on device (GPU)

CUDA introduces a syntax addition to C:
Triple angle brackets mark call from host code to device code. Contains organization and number of threads in two parameters:

myKernel<<< n, m >>>(arg1, …);

n and **m** will define organization of thread blocks, and threads in a block.

For now, we will set **n = 1**, which say one block and **m = N,** which says N threads in this block.

arg1, … , -- arguments to routine **myKernel** typically pointers to device memory obtained previously from **cudaMalloc**.

14

Declaring a Kernel Routine

Example – Adding to vectors A and B

```
#define N 256
__global__ void vecAdd(int *A, int *B, int *C) {  // Kernel definition

  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main() {
  // allocate device memory &
  // copy data to device
  // device mem. ptrs devA,devB,devC

  vecAdd<<<1, N>>>(devA,devB,devC);
  …
}
```

Each of the N threads performs one pair-wise addition:

```
Thread 0:   devC[0] = devA[0] + devB[0];
Thread 1:   devC[1] = devA[1] + devB[1];

Thread N-1: devC[N-1] = devA[N-1]+devB[N-1];
```

Grid of one block, block has N threads

15

5. Transferring data from device (GPU) to host (CPU)

Use CUDA routine cudaMemcpy

cudaMemcpy(&C, devC, size, cudaMemcpyDeviceToHost);

where **devC** is a pointer in device and **C** is a pointer in host.

16

6. Free memory in device and host

Use CUDA cudaFree routine:

cudaFree(devA);
cudaFree(devB);
cudaFree(devC);

free(a);
free(b);
free(c);

17

```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main (int argc, char **argv ) {

    int size = N *sizeof( int);
    int  *a,*b,*c, *devA, *devB, *devC;

    cudaMalloc( (void**)&devA, size) );
    cudaMalloc( (void**)&devB, size );
    cudaMalloc( (void**)&devC, size );

    a = (int*)malloc(size); b = (int*)malloc(size);c = (int*)malloc(size);

    cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( devB, b size, cudaMemcpyHostToDevice);

    vecAdd<<<1, N>>>(devA, devB, devC);

    cudaMemcpy( &c, devC size, cudaMemcpyDeviceToHost);
    cudaFree( devA;
    cudaFree( devB);
    cudaFree( devC);
    free( a ); free( b ); free( c );

    return (0);
}
```

18

Note about cudaMemcpy

- cudaMemcpy is synchronous
  - begins after all previous CUDA calls are complete
  - return control to host after copy is complete
- cudaMemcpyAsync: asynchronous version

```
// 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);
```

General grid/block specification

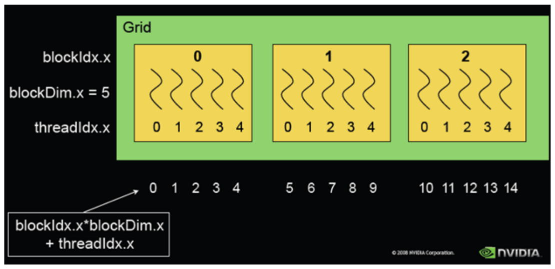
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Obtaining block ID: blockIdx.x, blockIdx.y

Obtaining thread ID: threadIDx.x, threadIdx.y, threadIdx.z

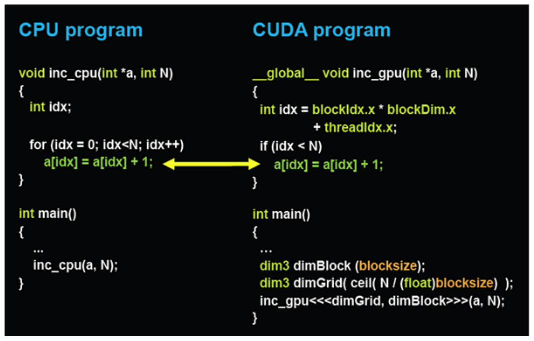
Obtaining dimensions of grid and block: gridDim, blockDim

Blocks and thread IDs



- Common programming pattern: map block-local threadIDs to a global thread ID
- Useful in many array programs for assigning work: see next slide

Using blockId and threadId



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);
}
```

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__    int LocalVar;`	local	thread	thread
`__device__ __shared__   int SharedVar;`	shared	block	block
`__device__              int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
  - Thread-local memory and spilled automatic variables is allocated in global memory

Synchronization

- **`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 or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

GPU Atomic Integer Operations

- Atomic operations on integers in global and shared memory:
  - Associative operations on signed/unsigned ints
    - add, sub, min, max, ...
    - and, or, xor
    - increment, decrement
  - Exchange, compare and swap
- Requires hardware with compute capability 1.1 and above.

25

Summary of C extensions

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
  - **gridDim, blockDim**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

26

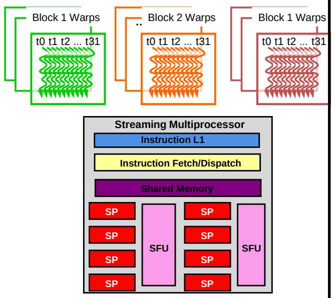
Performance Programming

Key ideas for performance

- Usual requirements
  - algorithm must have lot of parallelism
- Locality
  - use tiling to exploit shared-memory: copy blocks of data from global memory to shared-memory and operate on them
- New issue: exploiting SIMT
  - thread divergence: what happens when threads in a warp follow different control paths?
  - memory access optimization:
    - global memory: memory coalescing
    - shared memory: avoiding bank conflicts

Recall: Warp Scheduling

- Each Block is executed as 32-thread Warps
- Processor implements zero-overhead warp scheduling
  - any warp whose next instruction has its operands ready for consumption is eligible for execution
  - eligible warps are selected for execution on a prioritized scheduling policy
  - all threads in a warp execute the same instruction when selected



29

Thread divergence

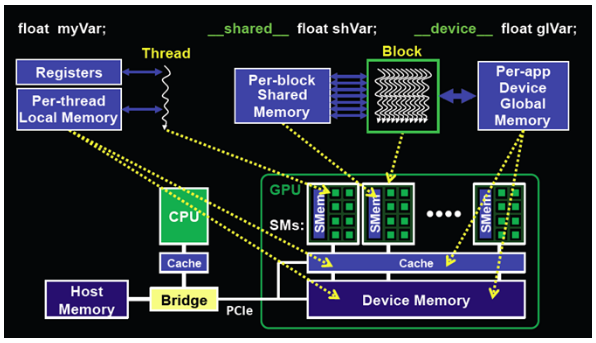
- Nothing in the programming model requires all threads within a warp to execute the same sequence of instructions!
- Conflict with SIMT: what happens if two threads within a warp want to take different execution paths?
  - common case: conditional branch which evaluates to different values in different threads
- Hardware: different execution paths are serialized
  - different control paths taken by the threads in a warp are traversed one at a time until they all converge again
- Thread divergence: performance problem at warp level

Avoiding thread divergence

- Example with divergence:
  - `If (threadIdx.x > 2) { }`
  - This creates two different control paths for threads in a block
  - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - `If (threadIdx.x / WARP_SIZE > 2) { }`
  - Also creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
- If possible, sort the work so that all the work done by a warp is similar (see example later)

31

Optimizing memory accesses



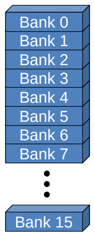
- Shared memory (on-chip): warp can send 32 distinct addresses in parallel
- Global memory (off-chip): one memory address per transaction

Optimizing memory accesses

- Optimizing accesses from a warp to shared and global memory is critical
- Different techniques needed
  - shared: avoid bank conflicts
  - global: memory coalescing
- Difference arises from architectural features
  - shared memory (on-chip): 32 different addresses can be presented in parallel from warp to memory
  - global memory (off-chip DRAM): only one address per memory bus transaction

Bank conflicts

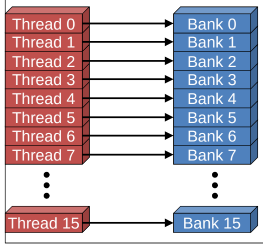
- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



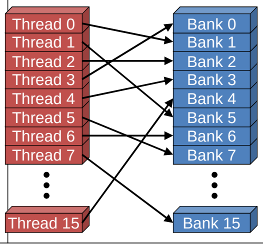
34

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1



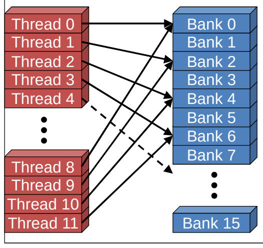
- No Bank Conflicts
  - Random 1:1 Permutation



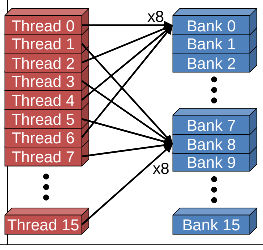
35

Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



36

How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

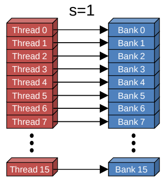
Linear Addressing

- Given:

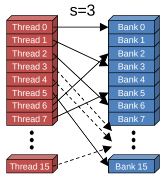
```
__shared__ float shared[256];
float foo =
  shared[baseIndex + s *
   threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so s must be odd

s=1



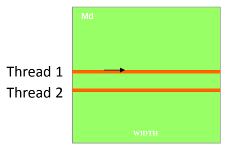
s=3



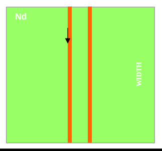
Optimizing global memory accesses

- Global memory (DRAM) is off-chip
  - only one address per memory transaction
  - each load transaction brings some number of aligned, contiguous bytes (say 32 Bytes) from memory (call them lines)
  - hardware automatically combines requests to same line from different threads in warp (coalescing)
  - multiple lines processed sequentially
- Optimization:
  - try to ensure that memory requests from a warp can be coalesced
  - (eg) stride-one access *across* threads in a warp is good (see picture)
  - (eg) use structure of arrays rather than array of structures

Not coalesced



Coalesced



Coalescing: detail

- Global memory requests from warp can be loads, stores or atomics
- Two or more threads in warp can specify the same address: what should happen?
- Load requests to same address: multicast
- Stores to same address: one thread will win
- Atomics to same address: serialize

CUDA programming examples

Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel
- Examples
  - Sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
- Typically parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

43

A Vector Reduction Example

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

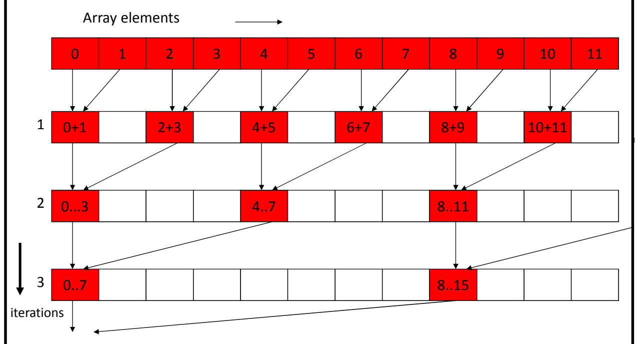
44

A simple implementation

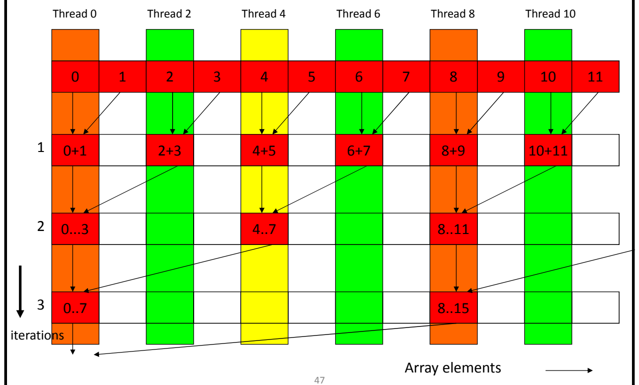
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
    stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

Vector Reduction with Bank Conflicts

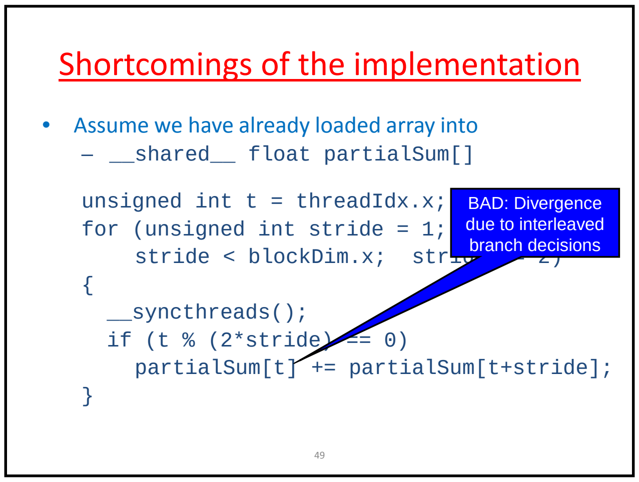


Vector Reduction with Branch Divergence



Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence
- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time
  - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence
    - This can go on for a while, up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire



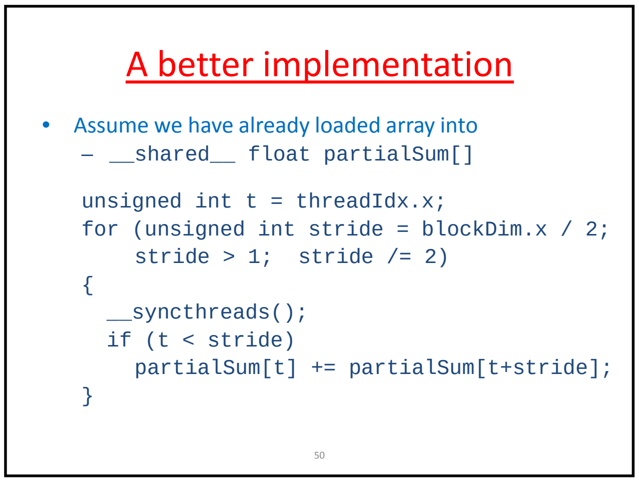
Shortcomings of the implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
    stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions

49



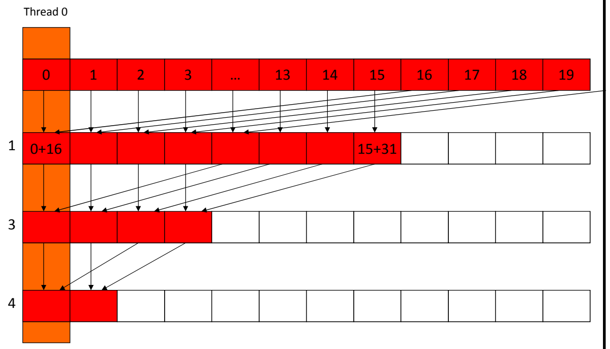
A better implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

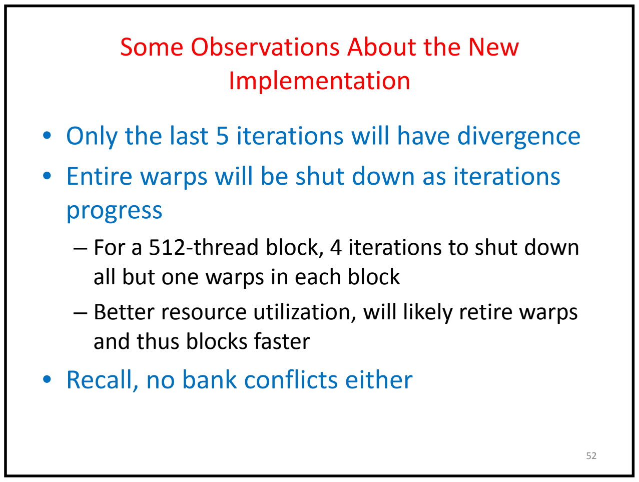
```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x / 2;
    stride > 1;  stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

50

No Divergence until < 16 sub-sums



51



Some Observations About the New Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either

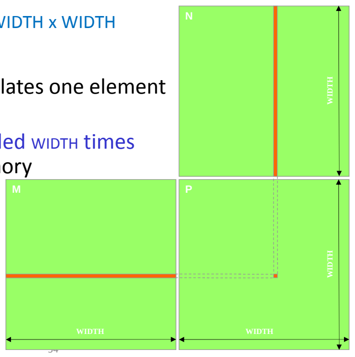
52

Matrix Multiplication

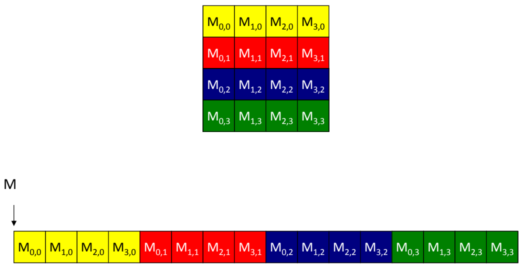
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Local, register and shared memory usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

Square Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory

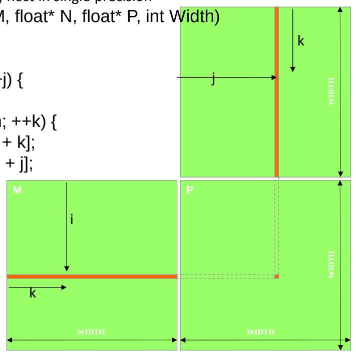


Memory Layout of a Matrix in C


$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Step 1: Matrix Multiplication A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in single precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;
   …
1. // Allocate and Load M, N to device memory
   cudaMalloc(&Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
   cudaMalloc(&Pd, size);
```

57

Step 3: Output Matrix Data Transfer (Host-side Code)

```
2.  // Kernel invocation code – to be shown later
    …

3.   // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

     // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

58

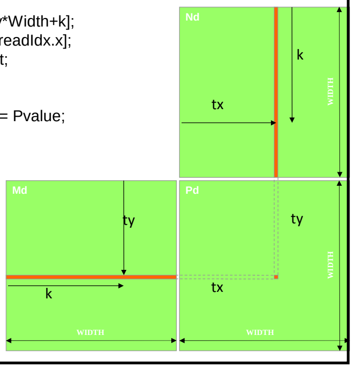
Step 4: Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

   // Pvalue is used to store the element of the matrix
   // that is computed by the thread
   float Pvalue = 0;
```

59

Step 4: Kernel Function (cont.)

```
 for (int k = 0; k < Width; ++k) {
     float Melement = Md[threadIdx.y*Width+k];
     float Nelement = Nd[k*Width+threadIdx.x];
     Pvalue += Melement * Nelement;
 }

 Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



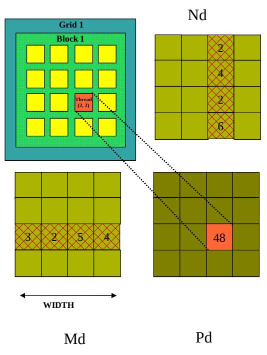
Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
   dim3 dimGrid(1, 1);
  dim3 dimBlock(Width, Width);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

61

Only One Thread Block Used

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Performs one multiply and one addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block

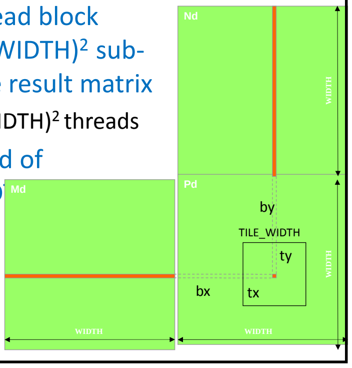


62

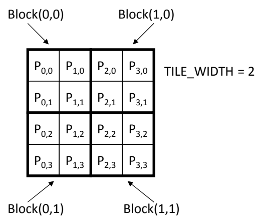
Step 7: Handling Arbitrary Sized Square Matrices

- Have each 2D thread block compute a (TILE_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (TILE_WIDTH)^2 threads
- Generate a 2D Grid of (WIDTH/TILE_WID

You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!



A Small Example



64

Revised Matrix Multiplication Kernel using Multiple Blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

65

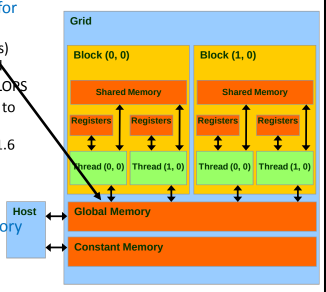
G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, because each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

66

How about performance on G80?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 4*346.5 = 1386 GB/s required to achieve peak FLOP rating
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



67

Matrix Multiplication using Shared Memory

68

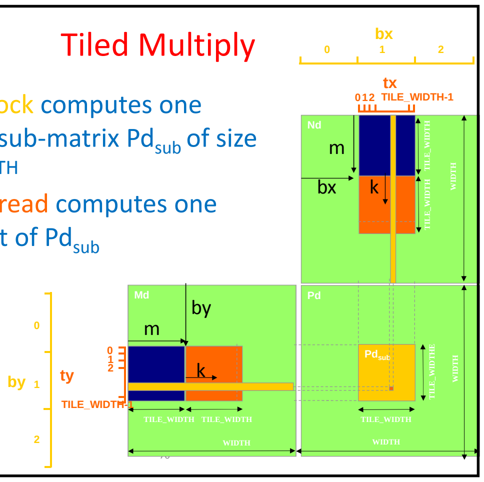
Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

     // Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.    float Pvalue = 0;
      // Loop over the Md and Nd tiles required to compute the Pd element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Coolaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.     Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.     __syncthreads();

11.     for (int k = 0; k < TILE_WIDTH; ++k)
12.       Pvalue += Mds[ty][k] * Nds[k][tx];
13.     __synchthreads();
14.   }
13.   Pd[Row*Width+Col] = Pvalue;
}
```

69

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



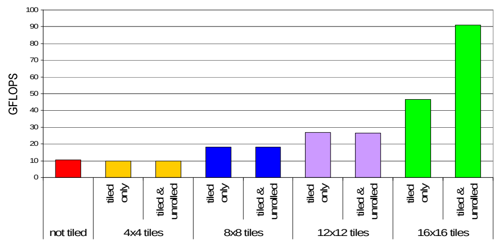
70

G80 Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing only up to two thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

71

Tiling Size Effects



72

GPU Applications

- Graphics
- Regular algorithms
  - CUBLAS, CUFFT, ….
- Barnes-Hut
  - Burtscher et al: NVIDIA GPU Gem
- Graph algorithms
  - Nasre et al
- See NVIDIA website for longer list

Summary

- Key features of GPUs
  - lots of threads: lightweight startup/shutdown
    - algorithms: should have lots of parallelism
  - warp-based execution: SIMT
    - not part of programming model
    - performance:
      - careful mapping of work to threads to avoid thread divergence
  - latency-tolerance: run lots of warps simultaneously to mask memory latency
  - exposed memory hierarchy managed by software
    - programming model: thread blocks and explicit data movement
    - performance:
      - memory coalescing: global memory
      - bank conflicts: shared-memory