The programming model

CPU program (serial code)

`cudaMemcpy ( ... )`

Copy data from CPU memory to GPU memory

Definition of a kernel (the function executed by each GPU thread)

`global_ Function ( ... )`

Launch a kernel with `nb` blocks, each with `nt` (up to 512) threads

`cudaMemcpy ( ... )`

Copy results from GPU memory to CPU memory

A kernel is a C function with the following restrictions:
- Cannot access host memory
- Must have “void” return type
- No variable number of arguments or static variables
- No recursion

The execution model

Blocks not dispatched initially are dispatched when some blocks finish execution

- The thread blocks are dispatched to SMs
- The number of blocks dispatched to an SM depends on the SM’s resources (registers, shared memory, …).

- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.
The execution model

- Each block (up to 512 threads) is divided into groups of 32 threads (called warps) – empty threads are used as fillers.
- The 32 threads of a warp execute in SIMD mode on the SM.
- The (up to 16) warps of a block are “coarse grain multithreaded”

- Depending on the number of SPs per SM:
  - If 32 SP per SM → one thread of a warp executes on one SP (32 lanes of execution, one thread per lane)
  - If 16 SP per SM → every two threads of a warp are time multiplexed (fine grain multithreading) on one SP (16 lanes of execution, 2 threads per lane)
  - If 8 SP per SM → every four threads of a warp are time multiplexed (fine grain multithreading) on one SP (8 lanes of execution, 4 threads per lane)

All threads execute the same code

Assume one block with 64 threads – launched using Kernel <<<1, 64>>>

```c
int i = threadIdx.x;
B[i] = A[63-i];
C[i] = B[i] + A[i]
```

- Each thread in a thread block has a unique “thread index” → threadIdx.x
- The same sequence of instructions can apply to different data
Blocks of threads

Assume two block with 32 threads each – launched using Kernel \( \llllll2, 32\rrrrrr \)

- Each thread block has a unique “block index” \( \text{blockIdx.x} \)
- Each thread has a unique \( \text{threadIdx.x} \) within its own block
- Can compute a global index from the \( \text{blockIdx.x} \) and \( \text{threadIdx.x} \)

\[
\text{int } i = 32 \times \text{blockIdx.x} + \text{threadIdx.x};
\]
\[
\text{B}[i] = \text{A}[63-i];
\]
\[
\text{C}[i] = \text{B}[i] + \text{A}[i]
\]

Two-dimensions grids and blocks

Can launch a 2x2 array of blocks, each consisting of 4x8 array of threads using Kernel \( \llllll(2,2), (4,8)\rrrrrr \)

- Each block has two indices (blockIdx.x, blockIdx.y)
- Each thread in a thread block has two indices (threadIdx.x, threadIdx.y)
Scheduling warps on SMs

Block diagram of a 16-lanes SM: A scoreboard keeps track of the current PCs (instruction address) of up to NW independent threads of SIMD instructions (NW warps). NW = 48 in older GPUs and 64 for more recent GPUs.

Up to 48 (64 in more recent GPUs) warps to be scheduled on the SM (may be from different blocks).

Scheduling warps on SMs

Scheduling threads of SIMD instructions (warps): The scheduler selects a ready instruction from some warp and issues it synchronously to all the SIMD lanes. Because warps are independent, the scheduler may select an instruction from a different warp at every issue.
SMs with multiple warp schedulers

Newer SMs have large number of SPs and may have multiple warp schedulers.

Example: SM with Dual warp Scheduler.
• Each SM has 32 SPs (cores) and is divided into two groups of 16 lanes each.
• One warp scheduler is responsible for scheduling a warp on one of the two 16-lanes groups.
• A warp is issued to each 16-lanes group every two cycles.

Sharing the resources of an SM

A single large register file (ex. 16K registers) is partitioned among the threads of the dispatched blocks.

A single SRAM (ex. 16KB) is partitioned among the dispatched blocks. All the threads of a block can access the partition assigned to that block.
The memory architecture

Can copy data between the CPU memory and the global memory using "cudaMemcpy()"

A variable allocated by the CPU using "cudaMalloc" or declared "_device_" in the kernel function is allocated in global memory and is shared by all the threads in the kernel.

A variable declared "_shared_" in the kernel function is allocated in shared memory and is shared by all the threads in a block.

Getting data into GPU memory

cudaMalloc(void **pointer, size_t nbytes); /* malloc in GPU global memory */
cudaMemset(void **pointer, int value, size_t count);
cudaMemcpy(void *dest, void *src, size_t nbytes, enum cudaMemcpyKind dir)
cudaFree(void **pointer);

enum cudaMemcpyKind
• cudaMemcpyHostToDevice
• cudaMemcpyDeviceToDevice
• cudaMemcpyDeviceToHost

Notes:
• cudaMemcpy() blocks CPU thread until copy is complete
• cudaMemcpy() does not start copying until previous CUDA calls complete
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(&a_d, nBytes);
  cudaMalloc(&b_d, nBytes);

  for (i=0; i<N; i++) a_h[i] = 100.f + i;
  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  GPUcomp<<<1, 14>>>(a_d, b_d, N);
  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;
}
```

Variable qualifiers in CUDA kernel functions

```
int main (void) {

  kernel <<<16,64 >>>

  ... ...

  ... ... 

  ... ... 

  ...

  _global_ void kernel( ... )
  {
    int i;
    _device_ float y[1024];
    _shared_ int x[64];

    x[threadIdx.x] = ...
    ... = y[blockIdx.x]
  }
```

GPU Global memory (DRAM)

One copy of the array x[] (64 elements) is allocated in shared memory for each thread block and can be accessed by each thread in that block.

y[] (1024 elements) is allocated in global memory and can be accessed by any thread (instead of using cudaMalloc).

One copy of i is allocated per thread – either in a register or in global memory (if run out of registers).
Function qualifiers

• The functional qualifier “_global_” is used to designate a kernel
  ➢ Called from Host and executed on Device
  ➢ Must return “void”

• The functional qualifier “_device_” designates a function
  ➢ Called from Device and executed on Device
  ➢ Cannot be called from Host

• The functional qualifier “_host_” designates a function
  ➢ Called from Host and executed on Host (default)

“_host_” and “_device_” can be combined to generate code that can execute on both Host and Device

Compiling

nvcc outputs either
➢ C code for the CPU
➢ PTX or object code for GPU

An executable requires linking to:
➢ “cudart” run time library
➢ “cuda” code library
Launching a kernel

- A kernel is launched using the syntax:
  \[ \text{kernel} \lll \text{dim3 dG, dim3 dB} \rrr (\ldots) ; \]

- Execution configuration:
  - \(dG\) → dimension and size of grids in blocks
  - 3-dimensions, \(dG.x, dG.y\) and \(dG.z\) (default \(dG.y = dG.z = 1\))
  - Number of blocks in the launched grid = \(dG.x \times dG.y \times dG.z\)
  - \(dB\) → dimension and size of blocks in threads
  - 3-dimensions, \(dB.x, dB.y\) and \(dB.z\) (default \(dB.y = dB.z = 1\))
  - Number of threads per block = \(dB.x \times dB.y \times dB.z\)

Note: kernel launch is non-blocking → control returns to CPU immediately
Example:

```c
void main ()
{
    cudaMalloc (int* &a, 20*sizeof(int));
    cudaMalloc (int* &b, 20*sizeof(int));
    cudaMalloc (int* &c, 20*sizeof(int));
    ...
    kernel<<<4,5>>>(a, b, c);
    ...
}

_global void kernel(int *a, *b, *c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    a[i] = i;
    b[i] = blockIdx.x;
    c[i] = threadIdx.x;
}
```

NOTE: Each block will consist of one warp – only 5 threads in the warp will do useful work and the other 27 threads will execute no-ops.

Example: increment the elements of an array

C program (on CPU)

```c
void inc_cpu(int *a, int n)
{
    int i;
    for(i=0; i<n; i++)
        a[i] = a[i] + 1;
}
```

CUDA program (on CPU+GPU)

```c
_global void inc_gpu(int *A, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if( i < n )
}

void main ()
{
    ...
    inc_cpu(a,n);
    ...
}
```

```c
void main ()
{
    ...n/64 64
    blocksize = 64;
    // cudaMalloc array A[n]
    // cudaMemcpy data to A
    dim3 dimB (blocksize);
    dim3 dimG(ceil(n/blocksize));
    inc_gpu<<<dimG, dimB>>>(A, n);
    ...
}
```
### Example: computing $y = ax + y$

<table>
<thead>
<tr>
<th>C program (on CPU)</th>
<th>CUDA program (on CPU+GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void saxpy_serial(int n, float a, float *x, float *y)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>for(int i = 0; i&lt;n; i++)</td>
<td></td>
</tr>
<tr>
<td>y[i] = a * x[i] + y[i];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>void main ()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>saxpy_serial(n, 2.0, x, y);</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>global_void saxpy_gpu(int n, float a, float *x, float *y)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int i = blockIdx.x*blockDim.x + threadIdx.x;</td>
<td></td>
</tr>
<tr>
<td>if (i &lt; n ) y[i] = a * x[i] + y[i];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void main ()</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>saxpy_serial(n, 2.0, x, y);</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

---

### Example: computing $y = ax + y$

```
_global void saxpy_gpu(int n, float a, float *X, float *Y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n ) Y[i] = a * X[i] + Y[i];
}
```

```
saxpy_gpu<<<4, 5>>>(18, 2.0, X, Y); /* X and Y are arrays of size 18 each */
```

---

May cause memory contamination.
Global Memory

- Global memory is the off-chip DRAM memory
  - Accesses must go through interconnect and memory controller
- Many concurrent threads generate memory requests → coalescing is necessary
  - Combining memory accesses made by threads in a warp into fewer transactions – each memory transaction is for 64 bytes (16, 4-bytes words).
  - E.g. if each thread of a warp are accessing consecutive 4-byte sized locations in memory, send two 64-byte request to DRAM instead of 32 4-byte requests
- Coalescing is achieved for any pattern of addresses that fits into a segment of size 128B.

Coalescing (cont.)

A warp may issue up to 32 memory accesses. They can be completed in
- Two transactions, each for 16 coalesced accesses (if perfectly coalesced)
- 32 separate transactions (if addresses cannot be coalesced)
- Fewer than 32 transactions (if partial coalescing is possible).
Shared Memory

- A memory address subspace in each SM (at least 48KB in nvidia gpus)
  - As fast as register files if no bank conflict
  - May be used to reduce global memory traffic (called scratchpad)
- Managed by the code (programmer)
- Many threads accessing shared memory → Highly banked
  - Successive 32-bit words assigned to successive banks
- Each bank serves one address per cycle
  - Shared Memory can service as many simultaneous accesses as it has banks
- Multiple concurrent accesses to a bank result in a bank conflict (has to be serialize)

Bank Addressing Example

- No Bank Conflicts
  - Linear addressing
  - Thread 0, 1, 2, 3, 4, 5, 6, 7 map to Bank 0, 1, 2, 3, 4, 5, 6, 7
- No Bank Conflicts
  - Random 1:1 Permutation
  - Slightly different mapping of threads to banks
Synchronization

- `_synchthreads_()`; a barrier for threads within a thread block
- Allowed in conditional code only if all condition is uniform in all the threads of a block
- Used to avoid hazards when using shared memory

To synchronize threads across different thread blocks, need to use **atomic operations** on variables in global memory

- Add, sub, min, max, …
- And, or, xor
- Increment, decrement
- Exchange, compare and swap

`cudaThreadSynchronize()`; called in CPU code to block until all previously issued cuda calls complete.
Maximize the use of shared memory

Shared memory is hundreds of times faster than global memory

To take advantage of shared memory
- Partition the data sets into subsets that fit into shared memory
- Handle each subset with one thread block
  - Load the subset from global memory to shared memory
  - __syncthreads()
  - Perform the computation while data is in shared memory
  - __syncthreads()
  - Copy results back from shared to global memory

The code examples in the next slides are copied and modified from:

Transposing an nxn matrix

• \( B[i][j] = A[j][i] \) for \( i=0,\ldots, n-1 \) and \( j=0,\ldots, n-1 \)
• Partitions \( A \) into \( k^2 = k \times k \) tiles, each with \( n/k \) rows and \( n/k \) columns
• Launch a 2-D grid of \( (k,k) \) thread blocks, each with \( (n/k,n/k) \) threads
• Assign \( (\text{threadIdx.x, threadIdx.y}) \) in \( (\text{blockIdx.x, blockIdx.y}) \) to copy element \( (\text{row, col}) \) of matrix \( A \) into position \( (\text{col, row}) \) of matrix \( B \).

\[
\begin{align*}
\text{global} & \quad \text{void transpose(float* A, float* B, int n)} \\
\{ & \quad \text{int row = blockIdx.y * blockDim.y + threadIdx.y;}
\quad \text{int col = blockIdx.x * blockDim.x + threadIdx.x;}
\quad \text{B[col][row] = A[row][col];} \\
\} \\
\end{align*}
\]

Naïve implementation exhibits non-coalesced access to global memory
__global__ void transpose(float* A, float* B, int n)
{
    _shared_ float C[tiledim][tiledim], D[tiledim][tiledim]; // defined parameter “tiledim” = n/k
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    C[threadIdx.y][threadIdx.x] = A[row][col]; // coalesced load from A
    _synchthreads() ; // complete loading before transposing
    D[threadIdx.x][threadIdx.y] = C[threadIdx.y][threadIdx.x] ;
    _synchthreads() ; // complete transpose in shared memory
    int row = blockIdx.x * blockDim.y + threadIdx.y; // note that blockDim.x = blockDim.y
    int col = blockIdx.y * blockDim.x + threadIdx.x;
    B[row][col] = D[threadIdx.y][threadIdx.x] // coalesced storing to B
}

Transposing with coalesced memory access

• Each block copies the rows of its tile from global to shared memory (coalesced)
• Transpose in shared memory
• Copy rows of the transposed tile from shared to global memory (coalesced)

Multiplication c = A * b of an nxn matrix, A, by a vector b

• Each element c[ij] will be computed by one thread
• Partitions c into k parts, each with n/k elements
• Launch k thread blocks, each with n/k threads
• Thread threadIdx.x in block blockIdx.x computes c[idx]

  where idx = blockIdx.x * blockDim.x + threadIdx.x

__global__ void mv(float* A, float* b, float* c, int n)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n) {
        float temp = 0 ;
        for (int k = 0; k < n; k++)
            temp += A[row][k] * b[k] ;
    }
    c[row] = temp ;
}

#define blocksize = 128; // or any other block size
n = 2048 ; // or any other matrix size
int nblocks = (n + blocksize - 1) / blocksize;
mv<<<nblocks, blocksize>>>(A, b, c, n);
Multiplication $C = A \times B$ of two $n \times n$ matrices.

- Partitions each matrix into $k^2 = k \times k$ tiles, each with $M$ rows and $M$ columns ($M = N/k$)
- Launch a 2-D grid of $(k,k)$ thread blocks, each with $(M,M)$ threads
- Assign thread (threadIdx.x, threadIdx.y) in block (blockIdx.x, blockIdx.y) to handle element (row, col) of the matrix C.

```c
__global__ void mm_simple(float* C, float* A, float* B, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int k = 0; k < N; k++)
        sum += A[row][k] * B[k][col];
    C[row][col] = sum;
}
```

Matrix-matrix multiplication using shared memory

- The thread that computes $C[\text{row}][\text{col}]$ accesses row $A[\text{row}][\text{*}]$ and column $B[\text{*}][\text{col}]$ from the global memory
- Hence, each row of $A$ and each column of $B$ is accessed $M$ times by different threads in the same block.
- To avoid repeated access to global memory, the block that computes a tile $C[i,j]$, where $0 \leq i < k$ and $0 \leq j < k$ executes:
  ```c
  for q = 0, … , k-1  // k = 3 in the example shown */
  {
    load tiles $A[i,q]$ and $B[q,j]$ into the shared memory
  }
  ```

40
Parallel reduction

```c
__global__ void reduce(int *input, int n, int *total_sum)
{
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ int x[blocksize];
    x[tid] = input[i]; // load elements into shared memory
    __syncthreads();
    // Tree reduction over elements of the block.
    for(int half = blockDim.x/2; half > 0; half = half/2)
    {
        if(tid < half) x[tid] += x[tid + half];
        __syncthreads();
    }
    // Thread 0 adds the partial sum to the total sum
    if( tid == 0 ) atomicAdd(total_sum, x[tid]);
}
```

```
#define blocksize = 128; // or any other block size
#define nblocks = n / blocksize;
reduce <<<nblocks, blocksize>>> (input, &total_sum);
```

Static Vs dynamic shared arrays

- When declaring a shared array using "__shared__ float C[tiledim][tiledim]", `tiledim` has to be statically defined. For example, in the matrix transpose example, the main should look like

```c
#define tiledim 16 ;      // blocksize = 16*16 = 256
n = 2048 ; // the dimension of the matrix
dim3 threads(tiledim, tiledim) ;
dim3 grid(n/tiledim, n/tiledim) ; // assuming n is a multiple of 16
transpose<<<grid, threads>>> (A, B, n) ;
```

- Can declare an unsized shared array using “extern __shared__ float C[]” (note the []) and then at kernel launch time, use a third argument to specify the size of the shared array:

```c
n = 2048 ;
tiledim = *** ; // can determine dynamically
dim3 threads(tiledim, tiledim) ;
dim3 grid(n/tiledim, n/tiledim) ;
transpose<<<grid, threads, tiledim*sizeof(float)>>>(A, B, n) ;
```
Dealing with data dependency

- Instructions in a warp are scheduled “in order”
- No pipeline forwarding → a dependent instruction cannot be issued
- Latency due to dependencies are hidden by issuing instructions from other warps (similar to multithreading – call it multi-warping)

```
add.f32  $f3, $f1, $f2
add.f32  $f5, $f3, $f4
ld.shared.f32 $f6, 0($r31)
add.f32  $f6, $f6, $f7
sw.shared.f32 $f6, 0($r31)
```

- Each SP is a 6-stage pipeline with no forwarding
- Two dependent instructions have to be separated by 6 other instructions
- Can completely hide the latency if the scheduler can issue instructions from 6 different warps (192 threads). They may be from different blocks.

---

Blocks per grid heuristics

The number of blocks should be larger than the number of SMs

The number of threads per SM is limited by the number of registers per SM – note that local variables not assigned to registers are stored in global memory. May use --maxregcount = n flag when compiling to restrict the number of registers per thread to n.

Example: If kernel uses 8 registers per thread, and register file is 16K registers per SM, then each SM can support at most 2048 threads)

The number of Blocks per SM is determined by the shared memory declared in each block and the number of threads per block.

Example: If 2KB shared memory is declared per block and each SM has 16KB of shared memory, then each SM can support at most 8 blocks). Example: If each block has 256 threads (8 warps), and the GPU can support 48 warps per SM, then each SM can support at most 6 blocks.
Unified CPU/GPU memory (in CUDA 6 and later)

- CPU and GPU share the same virtual memory (UVM)
- If physical memory is integrated (shared), then CPU and GPU use the same page table.
- If each of the CPU and the GPU has its own physical memory, pages are copied on demand (triggered by page faults).

Unified CPU/GPU memory

CPU code (not using a GPU)

```c
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}
```

CUDA 6 code with unified memory

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```

The pointer is passed as argument to the kernel → copying occurs on demand.

No need for

```c
malloc(&cpudata, N);
cudaMalloc(&data, N);
cudaMemcpy(&cpudata, &data ..);
```