CUDA

Sathish Vadhiyar

High Performance Computing

Hierarchical Parallelism

- Parallel computations arranged as grids
- One grid executes after another
- Grid consists of blocks
- Blocks assigned to SM. A single block assigned to a single SM. Multiple blocks can be assigned to a SM.
  - Max thread blocks executed concurrently per SM = 16

Hierarchical Parallelism

- Block consists of elements
- Elements computed by threads
  - Max threads per thread block = 1024
- A thread executes on a GPU core

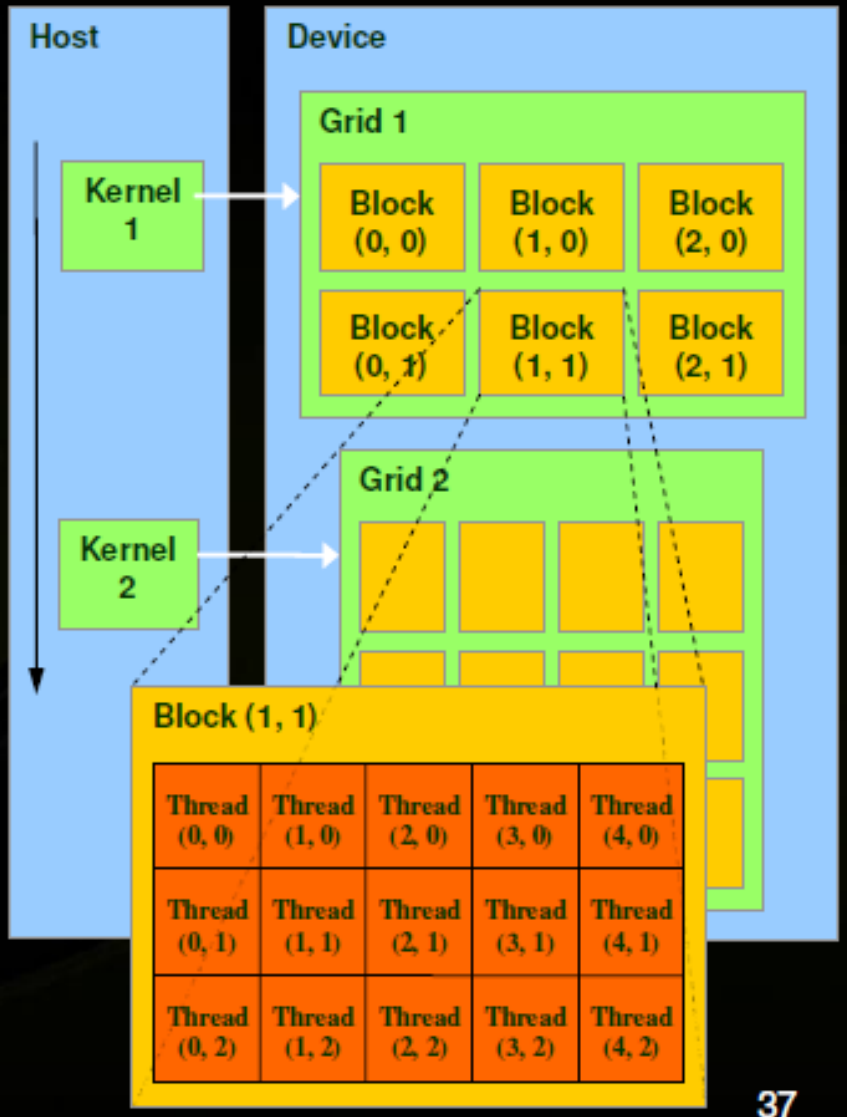
Thread Blocks

- Thread block – an array of concurrent threads that execute the same program and can cooperate to compute the result
- Has shape and dimensions (1d, 2d or 3d) for threads
- A thread ID has corresponding 1,2 or 3d indices
- Threads of a thread block share memory

CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



CUDA Programming Language

- Programming language for threaded parallelism for GPUs
- Minimal extension of C
- A serial program that calls parallel kernels
- Serial code executes on CPU
- Parallel kernels executed across a set of parallel threads on the GPU
- Programmer organizes threads into a hierarchy of thread blocks and grids

CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU; *Host* = CPU
Kernel = function that runs on the device

CUDA C

- Built-in variables:
  - threadIdx.{x,y,z} – thread ID within a block
  - blockIDx.{x,y,z} – block ID within a grid
  - blockDim.{x,y,z} – number of threads within a block
  - gridDim.{x,y,z} – number of blocks within a grid
- kernel<<<nBlocks,nThreads>>>(args)
  - Invokes a parallel kernel function on a grid of nBlocks where each block instantiates nThreads concurrent threads

Example: Summing Up

kernel function

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

grid of kernels

Variable Qualifiers (GPU code)

- __device__
  - stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - accessible by all threads
  - lifetime: application
- __constant__
  - same as __device__, but cached and read-only by GPU
  - written by CPU via cudaMemcpyToSymbol(...) call
  - lifetime: application
- __shared__
  - stored in on-chip shared memory (very low latency)
  - accessible by all threads in the same thread block
  - lifetime: kernel launch
- Unqualified variables:
  - scalars and built-in vector types are stored in registers
  - arrays of more than 4 elements stored in device memory

General CUDA Steps

1. Copy data from CPU to GPU
2. Compute on GPU
3. Copy data back from GPU to CPU

- By default, execution on host doesn't wait for kernel to finish
- General rules:
  - Minimize data transfer between CPU & GPU
  - Maximize number of threads on GPU

CUDA Elements

- cudaMalloc – for allocating memory in device
- cudaMemCopy – for copying data to allocated memory from host to device, and from device to host
- cudaFree – freeing allocated memory
- void syncthreads__() – synchronizing all threads in a block like barrier

EXAMPLE 1: MATRIX VECTOR MULTIPLICATION

Kernel

```
1  __global__ void matvec_mul(int m, int n, double *A, double *x,
       double *y)
2  {
3    int row, col;
4    double sum;
5
6    row =blockIdx.x*blockDim.x+threadIdx.x;
7
8    sum=0;
9    if(row < m){
10     for(col=0; col<n; col++){
11       sum += A[row*n+col]*x[col];
12     }
13   }
14   y[row]=sum;
15
16 }
17
```

Host Program

```
18 int main(int argc, char** argv){
19 ...
20 size_t size_A, size_x, size_y;
21 double *A, *x, *y;
22 double *dA, *dx, *dy;
23 ...
24
25   m = ... /* rows */; n = ... /* cols */
26
27   size_A = sizeof(double)*m*n;  size_x = sizeof(double)*n;
         size_y = sizeof(double)*m;
28
29   A = (double*)malloc(size_A);  x = (double*)malloc(size_x);  y =
          (double*)malloc(size_y);
30
31   /* Allocate on the device memory */
32   cudaMalloc((void **) &dA, size_A);
33   cudaMalloc((void **) &dx, size_x);
34   cudaMalloc((void **) &dy, size_y);
35
36   /* Initialize A and x */
37   /* Initialize y */
38   for(i=0; i<m; i++) y[i] = 0;
39
```

Host Program

```
40    /* Copy A and x to the device */
41    cudaMemcpy(dA, A, size_A, cudaMemcpyHostToDevice);
42    cudaMemcpy(dx, x, size_x, cudaMemcpyHostToDevice);
43
44    numThreadsPerBlock = 1024;   numBlocks = m/numThreadsPerBlock;
45
46    dim3 dimGrid(numBlocks);
47    dim3 dimBlock(numThreadsPerBlock);
48    matvec_mul<<< dimGrid, dimBlock >>>(m,n,dA,dx,dy);
49
50    cudaMemcpy(y, dy, size_y, cudaMemcpyDeviceToHost);
51
52    cudaFree(dA); cudaFree(dx); cudaFree(dy);
53
54    free(A); free(x); free(y);
55
56  }
```

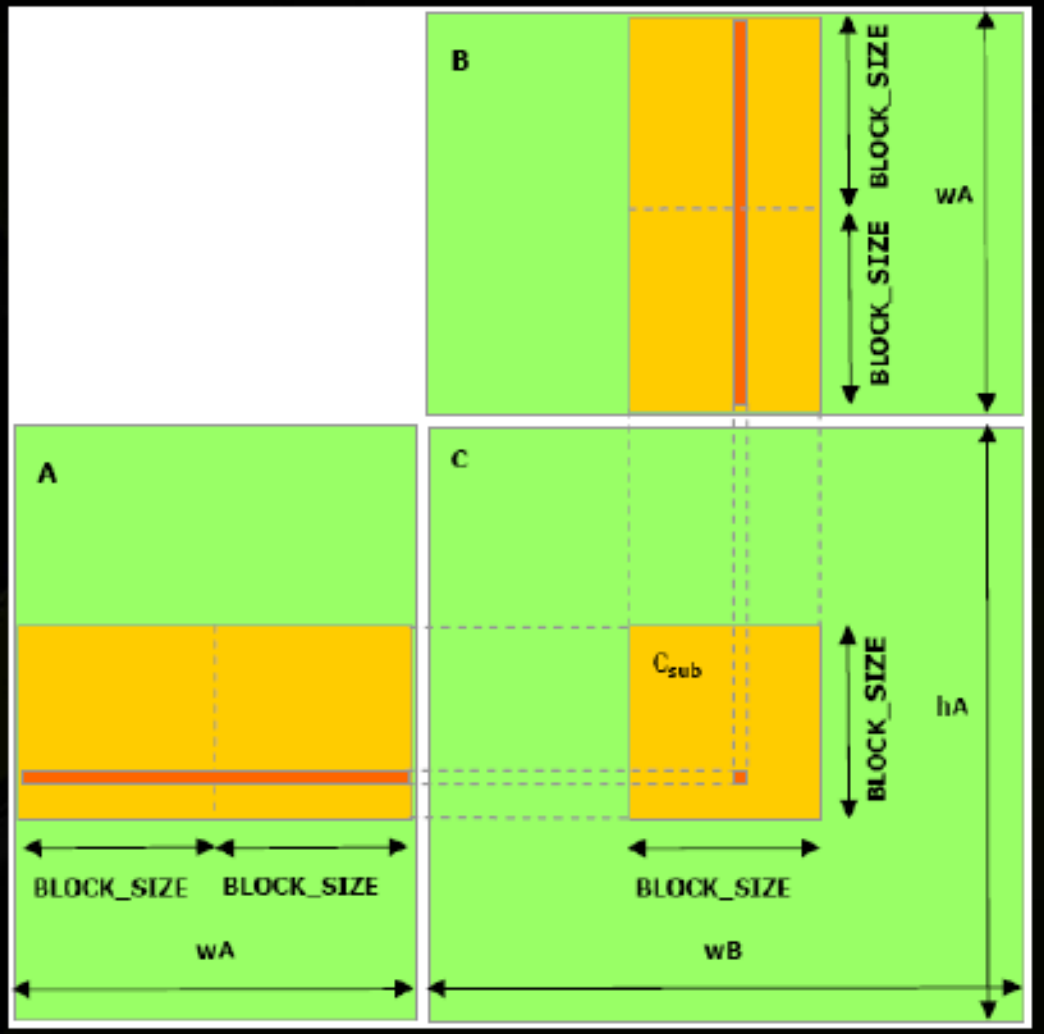
EXAMPLE 1, VERSION 2: ACCESS FROM SHARED MEMORY

```
1  __global__ void matvec_mul(int m, int n, double *A, double *x,
       double *y)
2  {
3    int row, col;
4    double sum;
5
6    __shared__ int sx[BLOCK_SIZE];
7
8    sx[threadIdx.x] = x[threadIdx.x];
9    __syncthreads();
0
1    row =blockIdx.x*blockDim.x+threadIdx.x;
2
3    sum=0;
4    if(row < m){
5      for(col=0; col<n; col++){
6        sum += A[row*n+col]*sx[col];
7      }
8    }
9    y[row]=sum;
0
1  }
```

EXAMPLE 2: MATRIX MULTIPLICATION

Matrix Multiplication Example

- Computing the product *C* of two matrices:
  - *A : (wA, hA)*
  - *B : (wB, wA).*
- Each thread block computes one square sub-matrix *Csub* of *C*;
- Each thread within the block computes one element of *Csub*.



Example 1: Matrix Multiplication

Host matrix multiplication code


```
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
  int size;
  // Load Input matrices A and B to the device
  float* Ad;
  size = hA * wA * sizeof(float);
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  .
  // Allocate memory for output matrix C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);
```

Example 1

```
    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
    // Read Ouput matrix C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(Ad);
    .
}
```

Example 1

Device matrix multiplication function


```
__global__ void Muld ( float* A, float* B, int wA, int wB, float* C)
{
  // Setup aBegin, aEnd, aStep    bBegin, bStep based on Block index and Block size

 // The element of the block sub-matrix that is computed by the thread
    float Csub = 0;
 // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
 for (int a = aBegin, b = bBegin;  a <= aEnd;  a += aStep, b += bStep) {

  // Shared memory for the sub-matrices of A and B
   __shared__  float As [ BLOCK_SIZE ] [ BLOCK_SIZE ];
   __shared__  float Bs [ BLOCK_SIZE ] [ BLOCK_SIZE ];
```

Example 1

```
// Load the matrices from global memory to shared memory; each thread loads one element of each matrix
As [ ty ] [ tx ] = A [ a + wA * ty + tx ];
Bs [ ty ] [ tx ] = B [ b + wB * ty + tx ];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together; each thread computes one element/ of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
     Csub += As[ty][k] * Bs[k][tx];
```

Example 1

```
    // Synchronize to make sure that the preceding computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }

  // Write the block sub-matrix to global memory; each thread writes one element
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

EXAMPLE 3: REDUCTION

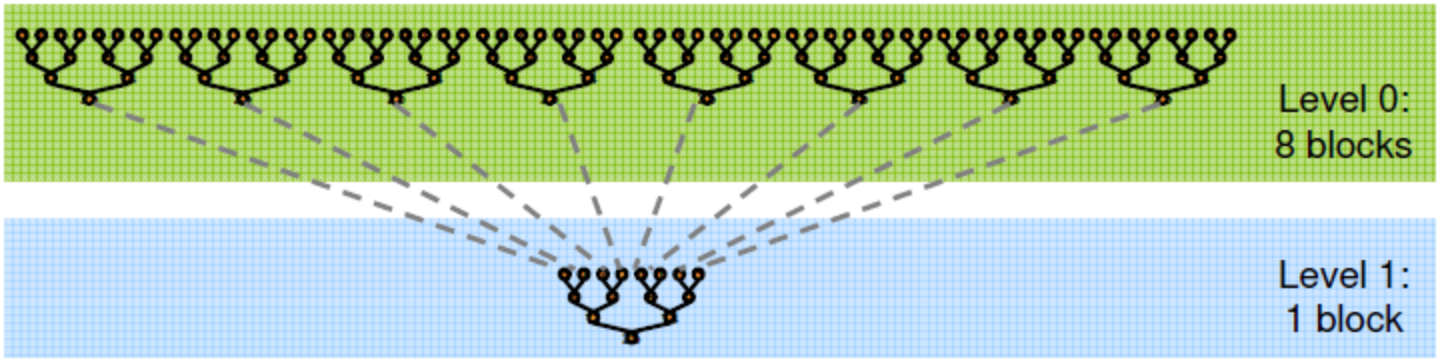
Example: Reduction

- Tree based approach used within each thread block
- In this case, partial results need to be communicated across thread blocks
- Hence, global synchronization needed across thread blocks

Reduction

- But CUDA does not have global synchronization –
  - expensive to build in hardware for large number of GPU cores
- Solution
  - Decompose into multiple kernels
  - Kernel launch serves as a global synchronization point

Illustration



Host Code

```
int main(){

int* h_idata, h_odata; /* host data*/
Int *d_idata, d_odata; /* device data*/

/* copying inputs to device memory */
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice) ;
cudaMemcpy(d_odata, h_idata, numBlocks*sizeof(int),
    cudaMemcpyHostToDevice) ;

int numThreadsperBlock = (n < maxThreadsperBlock) ? n : maxThreadsperBlock;
int numBlocks = n / numThreadsperBlock;
dim3 dimBlock(numThreads, 1, 1); dim3 dimGrid(numBlocks, 1, 1);
```

```
reduce<<< dimGrid, dimBlock >>>(d_idata, d_odata);
```

Host Code

```
int s=numBlocks;
while(s > 1) {
        numThreadsperBlock = (s< maxThreadsperBlock) ? s :
    maxThreadsperBlock;
        numBlocks = s / numThreadsperBlock;
        dimBlock(numThreads, 1, 1);     dimGrid(numBlocks, 1, 1);
         reduce<<< dimGrid, dimBlock, smemSize >>>(d_idata,
    d_odata);
         s = s / numThreadsperBlock;
}

}
```

Device Code

```
__global__ void reduce(int *g_idata, int *g_odata)
{
    extern __shared__ int sdata[];

    // load 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];
}
```

- For more information…
- CUDA SDK code samples – NVIDIA - http://www.nvidia.com/object/cuda_get_samples.html