An introduction to CUDA

Einstein Morales

University of Puerto Rico at Mayagüez

August 23, 2016

Contents

1. Introduction
2. CUDA execution model
3. CUDA threads
4. CUDA parallel programming
   Example
5. Questions

Introduction

CPU optimized for fast single-thread execution

- Cores designed to execute 1 thread or 2 threads concurrently
- Large caches attempt to hide DRAM access times
- Cores optimized for low-latency cache accesses

+

GPU optimized for high multi-thread throughput

- Cores designed to execute many parallel threads concurrently
- Cores optimized for data-parallel, throughput computation
- Chips use extensive multithreading to tolerate DRAM access times

CUDA: Support

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA
The CUDA platform is designed to work with programming languages such as C, C++ and Fortran.
CUDA: Compute Unified Device Architecture

- Fortran, Java, Python, C++ and others.
- MATLAB, Mathematica, R, LabView.

CUDA: Terminology

- Host: The CPU and its memory (host memory)
- Device: The GPU and its memory (device memory)

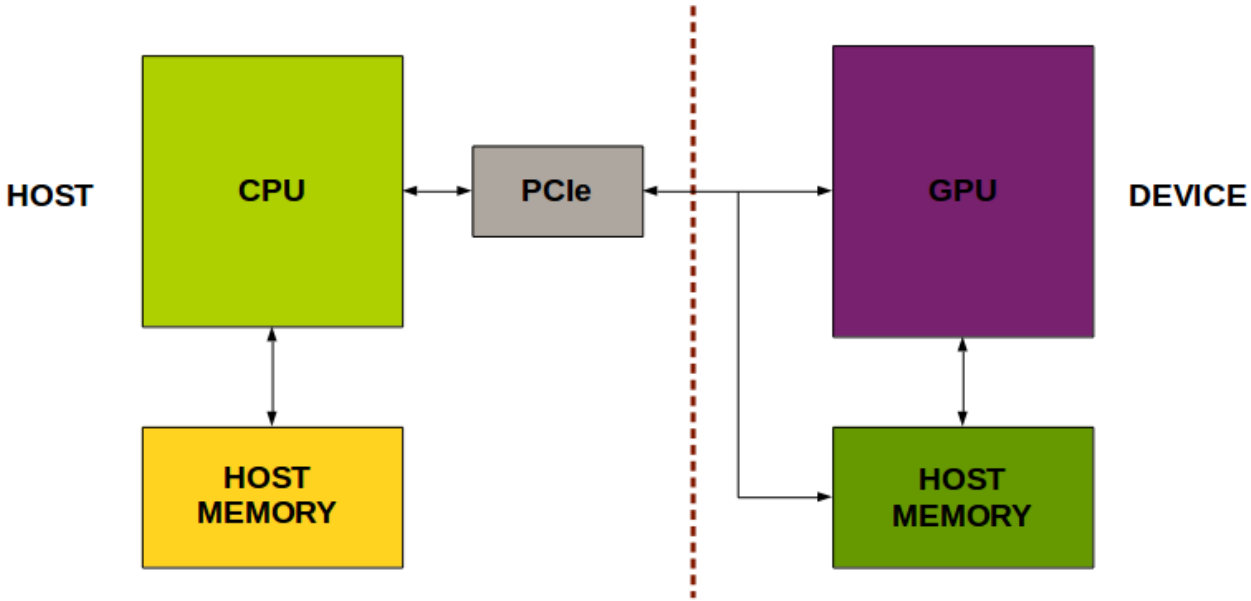


Fig. : CUDA execution flow

CUDA execution model

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements.

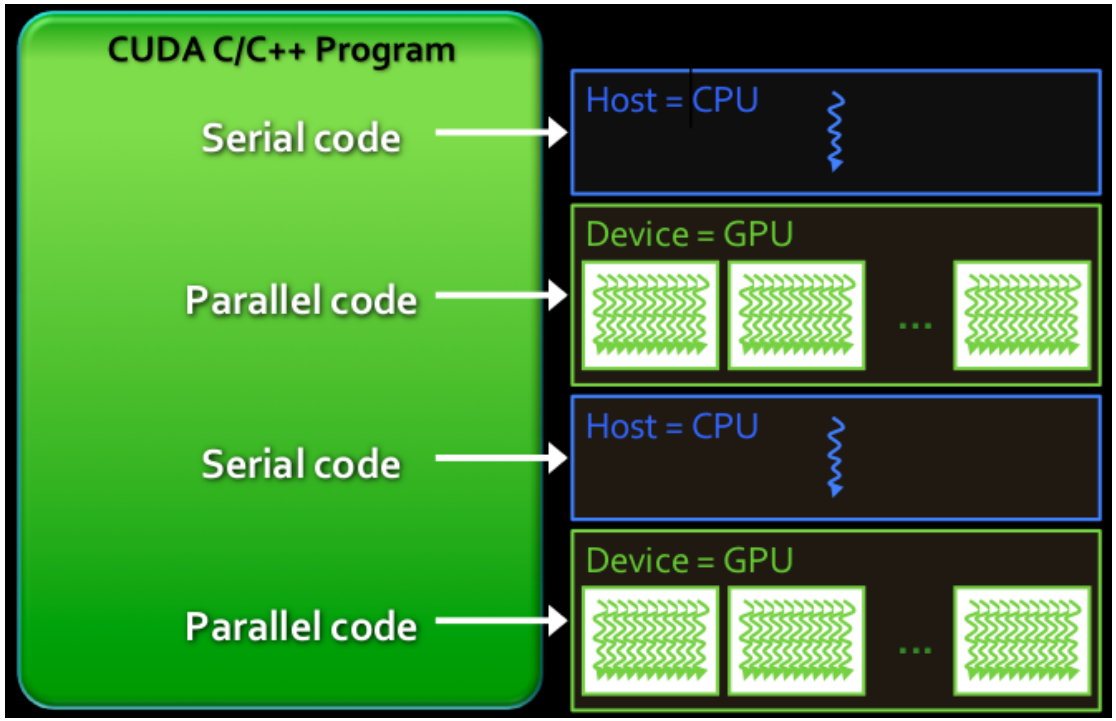


Fig. : CUDA execution model

CUDA Threads

- Difference between the threads of the CPU and GPU
  - GPU threads are very light
  - Thousands of threads are necessary for good performance, the CPU uses only a few.

CUDA Threads

- GPUs can handle thousands of concurrent threads.
- The pieces of code running on the gpu are called kernels
- A *kernel* is executed by a set of threads.
- All threads execute the same code (SPMD)
- Each thread has an index that is used to calculate memory addresses that this will access.

Example

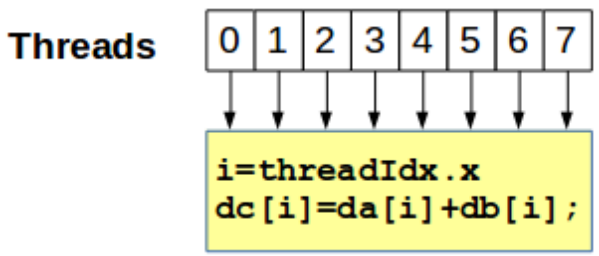
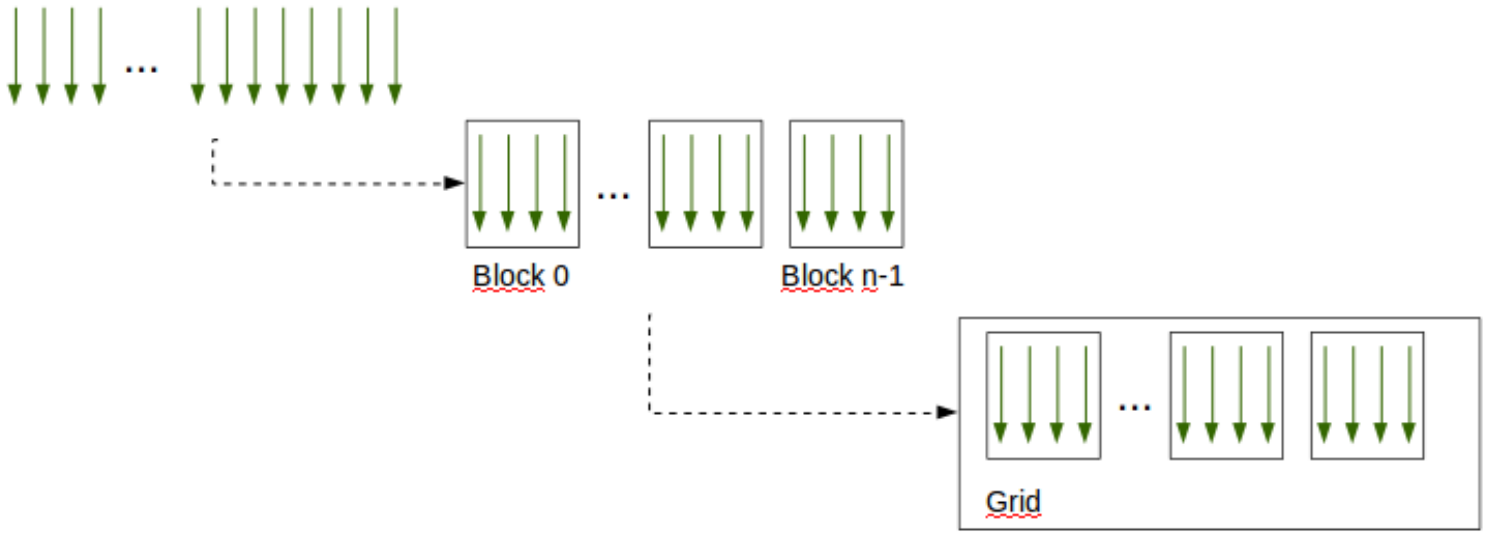


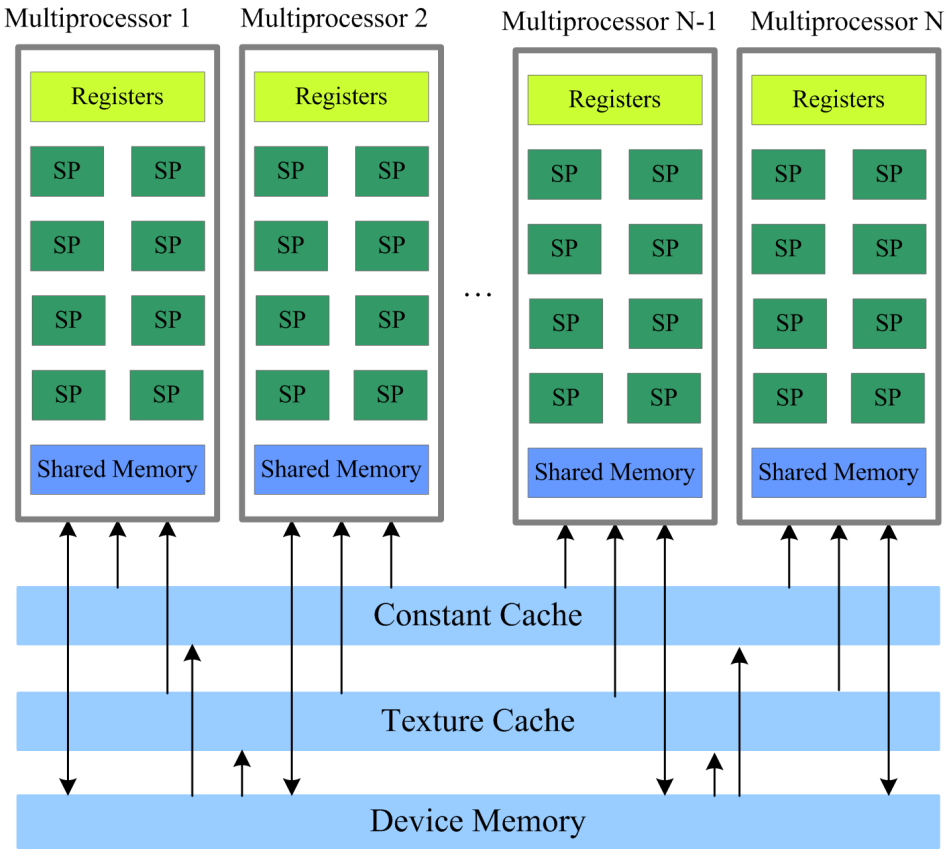
Fig. : thread identifier

CUDA threads organizations

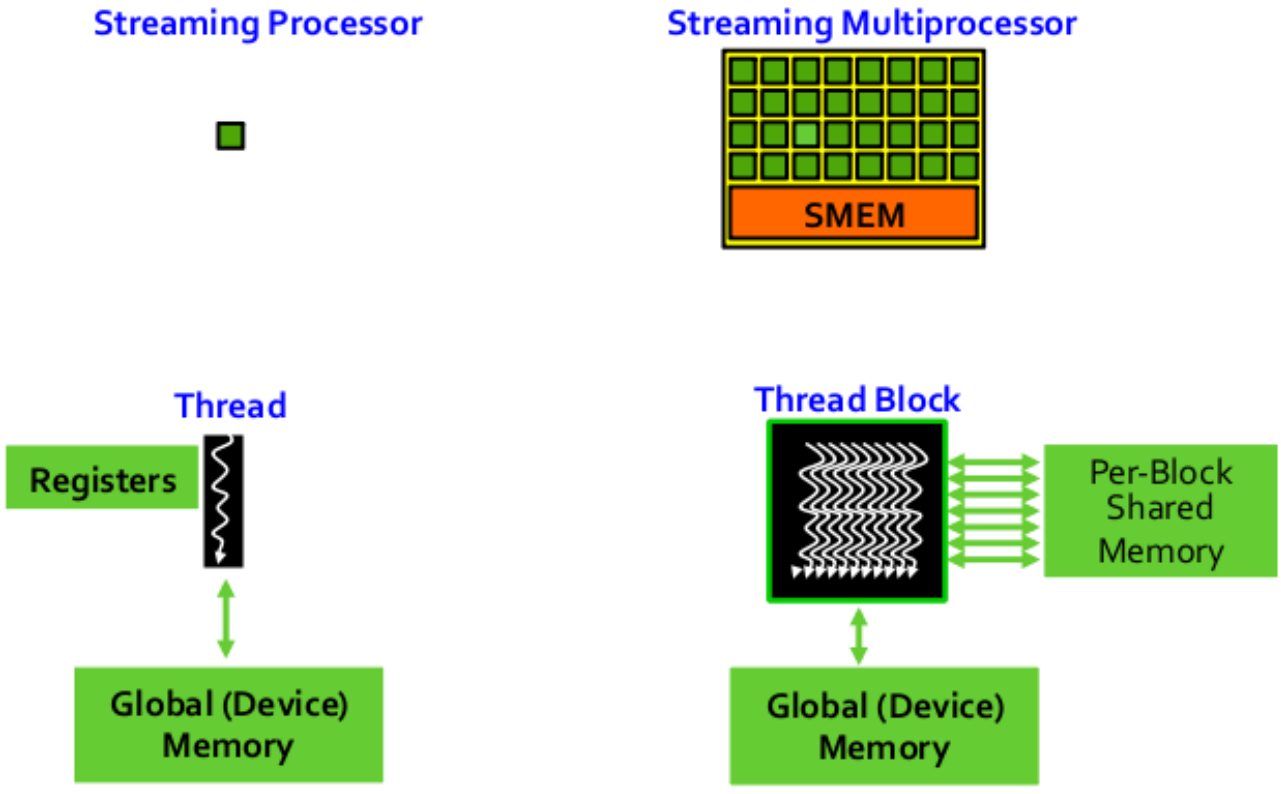


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A *kernel* is executed as a grid of blocks of threads

Streaming multiprocesors



Blocks execute on Streaming Multiprocessors

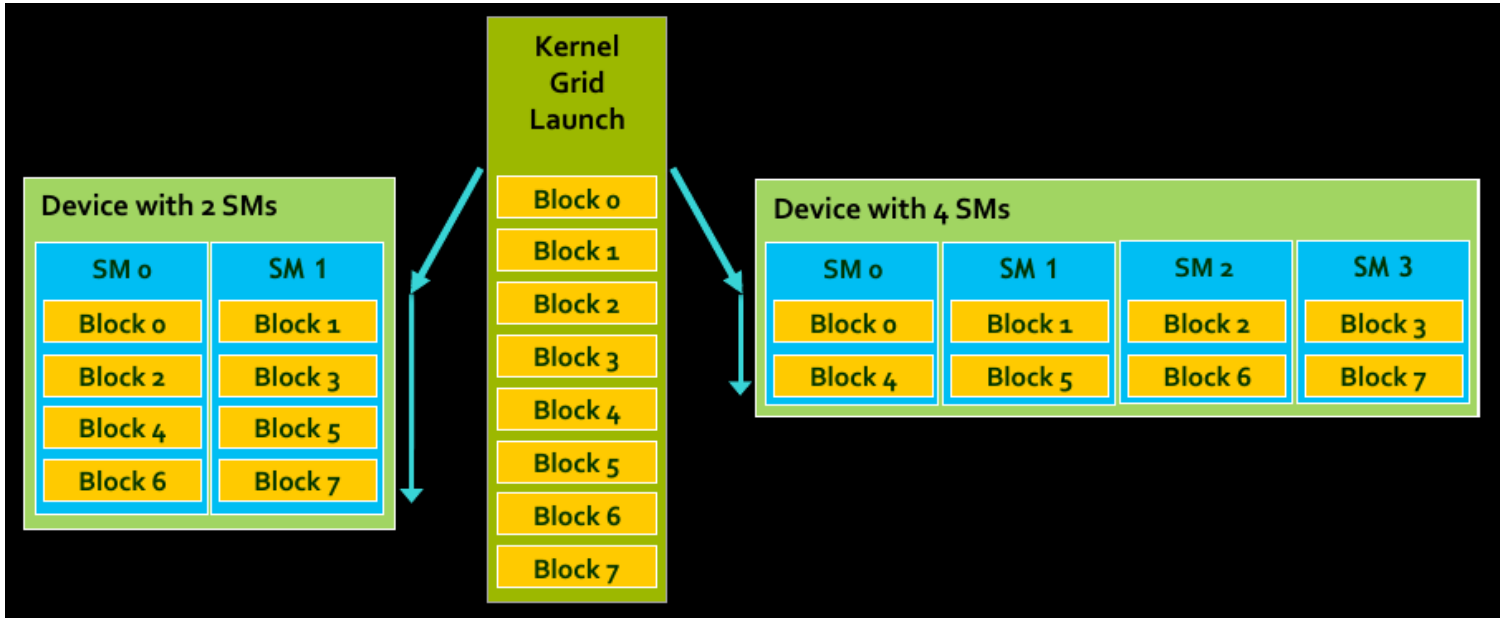


Kernel execution

1. A thread executes on a single SP.
2. A block executes on a single SM.
   - Threads and blocks do not migrate to different SMs.
   - All threads within block execute in concurrently, in parallel.
3. A SM may execute multiple blocks.
   - Must be able to satisfy aggregate register and memory demands.
4. A grid executes on a single device (GPU).

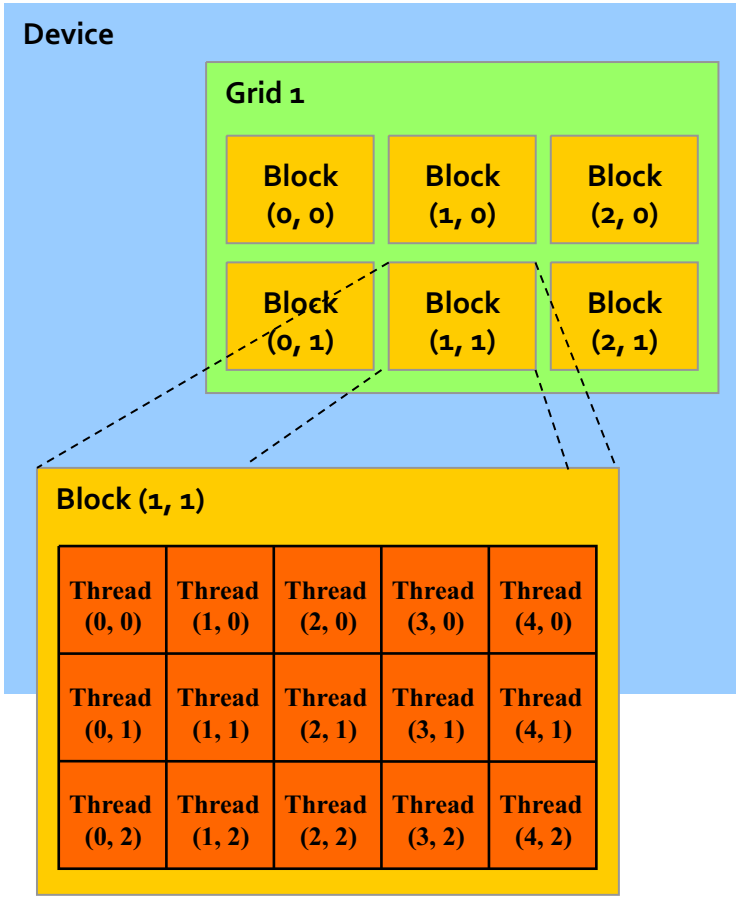
Independent execution of blocks provides scalability

Blocks can be distributed across any number of SMs



Thread and Block ID and Dimensions

- Built-in variables
  - `threadIdx`, `blockIdx`
  - `blockDim`, `gridDim`



Examples of Indexes and Indexing

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

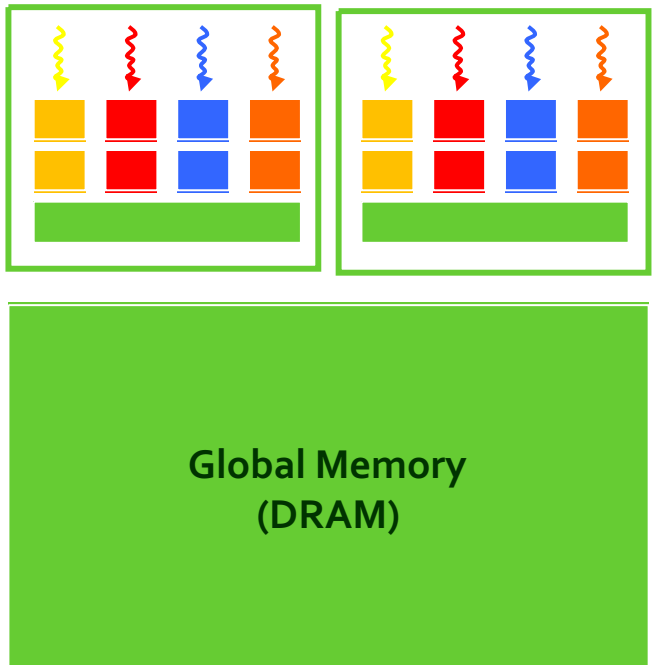
Example of 2D indexing

```
__global__ void kernel(int *a, int dimx, int
{
  int ix  = blockIdx.x*blockDim.x + threadIdx.x;
  int iy  = blockIdx.y*blockDim.y + threadIdx.y;
  int idx = iy * dimx + ix;

  a[idx]  = a[idx]+1;
}
```

CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global Memory



Example: SAXPY Kernel [1/4]

```
// [compute] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example: SAXPY Kernel [1/4]

```
// [computes] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Device Code

Example: SAXPY Kernel [1/4]

```
// [computes] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Host Code

```
int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [3/4]

```
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result…


  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y;

  return 0;
}
```

Example: SAXPY Kernel [3/4]

```
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result…

  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y;

  return 0;
}
```

Example: SAXPY Kernel [4/4]

```
void saxpy_serial(int n, float a, float* x, float* y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
// invoke host SAXPY function
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy(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];
}
// invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA C Code

questions so far