COMP 605: Introduction to Parallel Computing
Lecture : CUDA Block Parallelism

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 04/25/17
Last Update: 04/25/17

Table of Contents

CUDA Block Parallelism (K&W Ch3; S&K, Ch4)

Block Parallelism

- Simple add: CPU host launched a simple kernel that ran serially on the GPU device.
- Blocks: fundamental way that CUDA exposes parallelism: data parallelism
- Block parallelism will launch a device kernel that performs its computations in parallel.
- We will look at array addition:
  $add <<< N, 1 >>> (dev\_a, dev\_b, dev\_c)$;
- put multiple copies of the kernel onto the blocks

Serial CPU Code for Vector Add: add_loop_cpu.c

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 */
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Pseudocode for CPU and GPU Vector Add

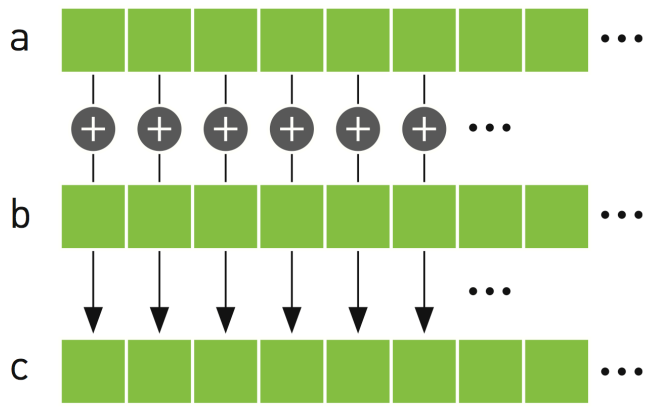
CPU

```
void add( int *a, int *b, int *c ) {
       int tid = 0;
       while (tid < N) {
           c[tid] = a[tid] + b[tid];
           tid += 2; }
}
. . . . . .
int main( void ) {
    . . . . . .
    add( a, b, c );
}
```

GPU

```
__global__ void add( int *a, int *b, int *c ) {
        int tid = blockIdx.x;
        if (tid < N)
            c[tid] = a[tid] + b[tid];
}
. . . . . .
int main( void ) {
    . . . . . .
    // set the number of parallel blocks
    // on device that will execute kernel
    // max number is 65,535 blocks
   add<<<N,1>>>( dev_a, dev_b, dev_c );
}
```

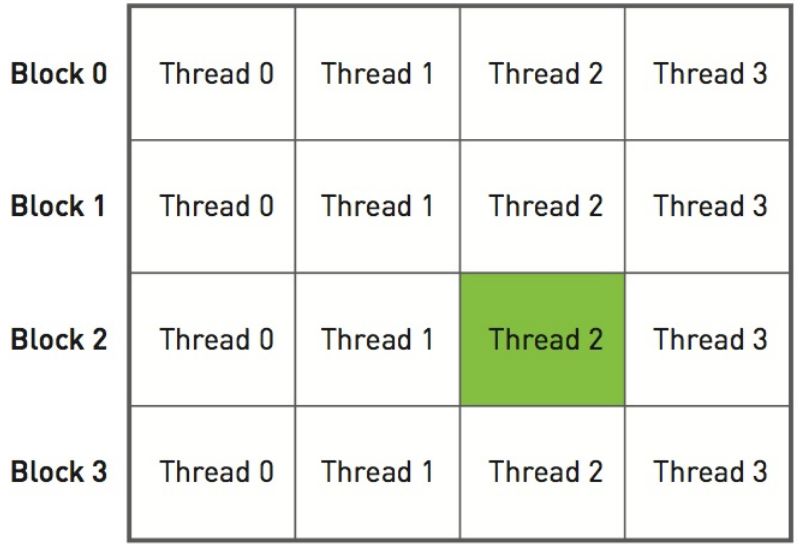
Summing two vectors



Recall: Grid and Block Assignment

kernelRoutine <<< *gridDim*, *blockDim* >>> (*args*)

- A *Grid* is a collection of blocks:
  - *gridDim*: size (dimensions) of grid of blocks
  - *blockIdx* : index (2D/3D indices) of block
- A *Block* is a collection of threads (columns):
  - *blockDim*: size (dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread
- *Threads* execute the *kernel* code on *device*:



Source: *Cuda By Example (Ch 5)*

GPU Code for the add kernel demonstrating how to obtain the block index ID.

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Example above is for GPU code with N blocks. Thread ID (or rank) is obtained from the block index object

Source: *Cuda By Example*

GPU block assignment for add kernel for $N = 4$ blocks, after *int tid = blockIdx.x* has been computed.

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

CUDA: Dynamic Variable Assignments

- The code below contains code for the simple add example from K&S
- It contains two key examples that show how to:
  - time your code using CUDA / *Event Timers*
  - pass variables from the command line and make them available to the device.
  - dynamically allocate memory on the host and device using dynamic variables.

```
/* File:   add_loop_gpu.cu
 *
 * Written By:   Mary Thomas (mthomas@mail.sdsu.edu)
 * Date:         Dec, 2014
 * Based on:     CUDA SDK code add_loop_gpu.cu
 * Description:  Reads number of threads from the command line
 *               and sets this as a global device variable.
 *
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement (EULA)
 * associated with this source code for terms and conditions that govern
 * your use of this NVIDIA software.
 *
 */

#include <stdio.h>

//#define N   65535+10
__device__ int d_Nthds;
 __global__ void checkDeviceThdCount(int *t) { *t = d_Nthds; }

__global__ void add( int *a, int *b, int *c) {
    int tid = blockIdx.x;    // this thread handles the data at its thread id
   if (tid < d_Nthds)
        c[tid] = a[tid] + b[tid];
}
```

```
int main( int argc, char** argv ) {
    int h_N = atoi(argv[1]);
    int a[h_N], b[h_N], c[h_N];
    int h_tid[h_N], h_cpuid[h_N];
    int *dev_a, *dev_b, *dev_c;
    int *d_tid;
    int i,j,k;
    int h_N2 = h_N;

    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

    // Check some key device properties
    int devCount=0;
    cudaGetDeviceCount(&devCount);
    printf("There are %d CUDA devices.\n", devCount);
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printf("Device Name: %s \n",  devProp.name);
        printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
        printf("Maximum dimensions of block: blockDim[0,1,2]=[");
        for (int i = 0; i < 3; ++i)
            printf(" %d ", devProp.maxThreadsDim[i]);
        printf("] \n");
    }
```

CUDA: Dynamic Variable Assignments

```
// set the number of threads to the global variable d_Nthds
  int h_Ndevice;
  int *d_N;
  cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int), 0, cudaMemcpyHostToDevice);
  cudaMalloc( (void**)&d_N, sizeof(int) );
  checkDeviceThdCount<<<1,1>>>(d_N);
   cudaMemcpy( &h_Ndevice, d_N, sizeof(int), cudaMemcpyDeviceToHost ) ;
  printf("h_N = %d, h_Ndevice=%d \n", h_N, h_Ndevice);
  cudaThreadSynchronize();


  // allocate the memory on the GPU
  cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
  cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
  cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;

  // fill the arrays 'a' and 'b' on the CPU
  for (i=0; i<h_N; i++) {
      a[i] = i+1;
      b[i] = (i+1) * (i+1);
  }

  // copy the arrays 'a' and 'b' to the GPU
   cudaMemcpy( dev_a, a, h_N * sizeof(int), cudaMemcpyHostToDevice ) ;
   cudaMemcpy( dev_b, b, h_N * sizeof(int), cudaMemcpyHostToDevice ) ;

  //getThreadInfo<<<h_N,1>>>( d_tid,d_cpuid );
  add<<<h_N,1>>>( dev_a, dev_b, dev_c);

  // copy the array 'c' back from the GPU to the CPU
  cudaMemcpy( c, dev_c, h_N * sizeof(int), cudaMemcpyDeviceToHost ) ;
```

```
    //print out small arrays
    if( h_N < 11 )
    {
        // display the results
        for (i=0; i<h_N; i++) {
            printf( "%d + %d = %d\n", a[i], b[i], c[i] );
        }

       i=0; j=h_N/2; k=h_N-1;
       printf( "Arr1[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
       printf( "Arr1[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
       printf( "Arr1[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );

       printf( "Arr2[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
       printf( "Arr2[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
       printf( "Arr2[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );
    }

    // free the memory allocated on the GPU
     cudaFree( dev_a ) ;
     cudaFree( dev_b ) ;
     cudaFree( dev_c ) ;

    //calculate elapsed time:
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    //Computes the elapsed time between two events (in milliseconds)
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);

    return 0;
}
```

Batch Script for Running CUDA add_loop_gpu.cu

```
#!/bin/sh
#
#  to run:
#        qsub -v T=10 bat.addloop
#
#PBS -V
#PBS -l nodes=1:node9:ppn=1
#PBS -N addloop
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR

echo "running add_loop_gpu using $T threads"
./add_loop_gpu $T
```

add_loop_gpu.cu (output), also showing device information and timing diagnostics

```
[mthomas@tuckoo chapter04]$qsub -v NTHDS=10 bat.addloop
running add_loop_gpu using 10 threads
There are 2 CUDA devices.

CUDA Device #0
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]

CUDA Device #1
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
h_N = 10, h_Ndevice=10
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
9 + 81 = 90
10 + 100 = 110
Arr1[0]: 1 + 1 = 2
Arr1[5]: 6 + 36 = 42
Arr1[9]: 10 + 100 = 110
Arr2[0]: 1 + 1 = 2
Arr2[5]: 6 + 36 = 42
Arr2[9]: 10 + 100 = 110
GPU Nthreads=10, Telap(msec)=         0.5985599756240845
```

Timing CUDA code CudaEvent Timers (output)

```
int main( int argc, char** argv ) {
    . . .
    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    . . .
    cudaEventRecord(start, 0) ;
    . . .
    . . .
    //calculate elapsed time:
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    //Computes the elapsed time between two events (in milliseconds)
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);
```

See S&K, Chapter 6

Timing Results for add_vector output using CudaEvent Timers (output)

```
serial: Nthreads=10000,      Telap(msec) =       184.0
serial: Nthreads=1000000,    Telap(msec) =     15143.0
serial: Nthreads=100000000,  Telap(msec) =    181107.0


   GPU: Nthreads=10000,      Telap(msec) =        1.1845
   GPU: Nthreads=1000000,    Telap(msec) =        11.185
   GPU: Nthreads=100000000,  Telap(msec) =        661.78
```

What happens to global variables?

Set up test code to see results for very large values of N:

```
    i=0;  j=N/2;  k=N;
    printf( "Arr1[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
    printf( "Arr1[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
    printf( "Arr1[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );

    i=0;  j=N2/2;  k=N2;
    printf( "Arr2[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
    printf( "Arr2[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
    printf( "Arr2[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );
```

```
Arr1[0]: 0 + 0 = 32896
Arr1[65540]: 65540 + 524304 = 0
Arr1[65545]: 11028 + 0 = 0
Arr2[0]: 0 + 0 = 32896
Arr2[32772]: 32772 + 1074003984 = 0
Arr2[65545]: 11028 + 0 = 0
```

What happens when the number of threads is >> number of blocks?

- Need to distribute the threads
- Cannot exceed *maxThreadsPerBlock*, typically 512
- Need a combination of threads and blocks
- Algorithm to convert from 2D space (multiple blocks and multiple threads per block) to 1D:

  $int \quad tid = threadIdx.x + blockIdx.x * blockDim.x;$
- Note: *blockDim* is constant

add.cu

```
#include "../common/book.h"

#define N   (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a,N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b,N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c,N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
```

```
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<128,128>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i]
            success = false;
        }
    }
    if (success)    printf( "We did it!\n" );

    // free the memory we allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    // free the memory we allocated on the CPU
    free( a );   free( b );    free( c );

    return 0;
    }
```

Two dimensional arrangement of a collection of blocks and threads

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Source: *Cuda By Example, Figure 5.1*

CUDA Hello World: Using dimGrid and dimBlock

```
#include <stdio.h>
__global__ void hello_kernel(float * x) {
// By Ingemar Ragnemalm 2010
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)  {
        a[threadIdx.x] += b[threadIdx.x];
}

int main()  {
        char a[N] = "Hello \0\0\0\0\0\0";
        int b[N] = {15, 10, 6, 0, -11, 1, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0};

        char *ad;
        int *bd;
        const int csize = N*sizeof(char);
        const int isize = N*sizeof(int);

        printf("%s", a);

        cudaMalloc( (void**)&ad, csize );
        cudaMalloc( (void**)&bd, isize );

        cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
        cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

}
```

```
        dim3 dimBlock( blocksize, 1 );

        dim3 dimGrid( 1, 1 );

        hello<<<dimGrid, dimBlock>>>(ad, bd);

        cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );

        cudaFree( ad );   cudaFree( bd );

        printf("%s\n", a);
        return EXIT_SUCCESS;
```