Introduction to programming in CUDA C

Will Landau

Iowa State University

September 30, 2013

Outline

Outline

The single instruction, multiple data (SIMD) paradigm

- SIMD: apply the same command to multiple places in a dataset.

```
for(i = 0; i < 1e6; ++i)
  a[i] = b[i] + c[i];
```

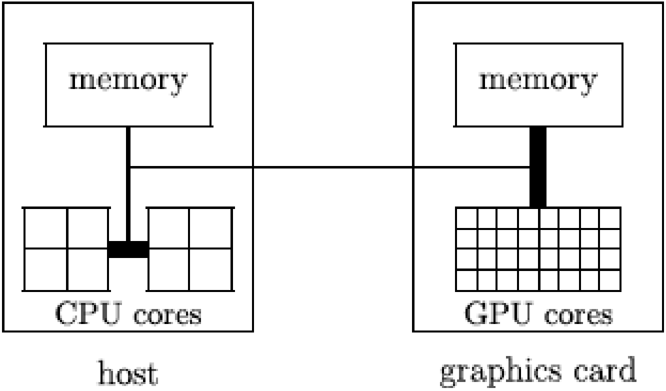
- On CPUs, the iterations of the loop run sequentially.
- With GPUs, we can easily run all 1,000,000 iterations simultaneously.

```
i = threadIdx.x;
a[i] = b[i] + c[i];
```

- We can similarly *parallelize* a lot more than just loops.

CPU / GPU cooperation

- The CPU ("host") is in charge.
- The CPU sends computationally intensive instruction sets to the GPU ("device") just like a human uses a pocket calculator.



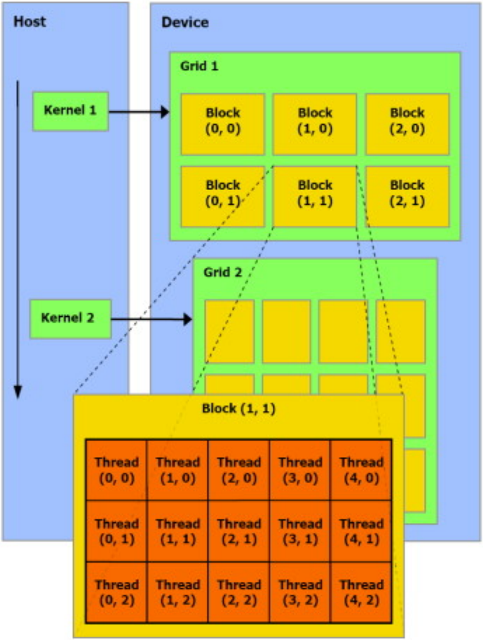
How GPU parallelism works

1. The CPU sends a command called a **kernel** to a GPU.
2. The GPU executes several duplicate realizations of this command, called **threads**.
   - These threads are grouped into bunches called **blocks**.
   - The sum total of all threads in a kernel is called a **grid**.

- Toy example:
  - CPU says: "Hey, GPU. Sum pairs of adjacent numbers. Use the array, (1, 2, 3, 4, 5, 6, 7, 8)."
  - GPU thinks: "Sum pairs of adjacent numbers" is a kernel.
  - The GPU spawns 2 blocks, each with 2 threads:

Block	0		1	
Thread	0	1	0	1
Action	$1 + 2$	$3 + 4$	$5 + 6$	$7 + 8$

- I could have also used 1 block with 4 threads and given the threads different pairs of numbers.



Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

CUDA: making a gaming toy do science

- **CUDA**: Compute Unified Device Architecture.
- Before CUDA, programmers could only do GPU programming in graphics languages, which are appropriate for video games but clumsy for science.
- CUDA devices support CUDA C, an extension of C for programs that use GPUs.
- CUDA-enabled servers at Iowa State:
  - impact1.stat.iastate.edu
  - impact2.stat.iastate.edu
  - impact3.stat.iastate.edu
  - impact4.stat.iastate.edu (in the works...)

Outline

Hello world

- A beginner C program:

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
  return 0;
}
```

- A beginner CUDA C program:

```
#include <stdio.h>

__global__ void myKernel(){
}

int main(){
  myKernel<<<1, 1>>>();
  printf("Hello, World!\n");
  return 0;
}
```

Hello world

```c
#include <stdio.h>

__global__ void myKernel(){
}

int main(){
  myKernel<<<2, 4>>>();
  printf("Hello, World!\n");
  return 0;
}
```

- `__global__` says that the function is a kernel, which
  - will be executed on the GPU by one or more simultaneous threads when called.
  - must return void
- `<<<2, 4>>>` specifies
  - number of blocks (first number)
  - number of threads per block (second number).

Prefixes in CUDA C

- `__host__`
  - Runs once per call on the CPU.
  - Only callable from the CPU (i.e., from another host function).
  - All functions without explicit prefixes are host functions.
- `__global__`
  - Used to specify a kernel.
  - Runs multiple times per call on the GPU (that's what `<<<#, #>>>` is for).
  - Only callable from the CPU (i.e., from a host function).
- `__device__`
  - Runs once per call on the GPU.
  - Only callable from the GPU (i.e., from either a kernel or another device function).

Prefix example: 2 blocks and 5 threads per block

```
#include <stdio.h>

__device__ int dev1(){
}

__device__ int dev2(){
}

__global__ void pleaseRunThis10Times(){
  dev1();
  dev2();
}

int main(){
  pleaseRunThis10Times<<<2, 5>>>();
  printf("Hello, World!\n");
  return 0;
}
```

skeleton.cu: outlining a CUDA C workflow

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void some_kernel(...){...}

int main (void){
  // Declare all variables.
  ...
  // Allocate host memory.
  ...
  // Dynamically allocate device memory for GPU results.
  ...
  // Write to host memory.
  ...
  // Copy host memory to device memory.
  ...

  // Execute kernel on the device.
  some_kernel<<< num_blocks, num_theads_per_block >>>(...);

  // Write GPU results in device memory back to host memory.
  ...
  // Free dynamically-allocated host memory
  ...
  // Free dynamically-allocated device memory
  ...
}
```

simple.cu: a program that actually does something

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void colonel(int *a_d){
  *a_d = 2;
}
int main(){
  int a = 0, *a_d;

  cudaMalloc((void**) &a_d, sizeof(int));
  cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

  colonel<<<1,1>>>(a_d);

  cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);

  printf("a = %d\n", a);
  cudaFree(a_d);

}
```

Compiling and running simple.cu

```
> nvcc simple.cu -o simple
> ./simple
a = 2
```

- Notes:
  - nvcc is the NVIDIA CUDA C compiler,
  - CUDA C source files usually have the *.cu extension, though they sometimes have have *.c and *.cpp extensions.
  - This code is available at http://will-landau.com/gpu/Code/CUDA_C/simple/simple.cu.
  - Most of the example code I present will be linked from pages at will-landau.com/gpu/talks.

Builtin CUDA C variables

- `maxThreadsPerBlock`: exactly that: 1024 on impact1.
- For a kernel call with B blocks and T threads per block,
  - `blockIdx.x`
    - ID of the current block (in the x direction).
    - Integer from 0 to $B - 1$ inclusive.
  - `threadIdx.x`
    - within the current block, ID of the current thread (in the x direction).
    - Integer from 0 to $T - 1$ inclusive.
  - `gridDim.x`: number of blocks in the current grid (in the x direction).
  - `blockDim.x`: number of threads per block (in the x direction).
- With some modifications that I will describe in later lectures, you can use the y and z directions with variables like `threadIdx.y`, `threadIdx.z` etc.

Vector addition: `vectorsums.cu`

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N 10

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

int main(void) {
  int i, a[N], b[N], c[N];
  int *dev_a, *dev_b, *dev_c;

  cudaMalloc((void**) &dev_a, N*sizeof(int));
  cudaMalloc((void**) &dev_b, N*sizeof(int));
  cudaMalloc((void**) &dev_c, N*sizeof(int));

  for(i=0; i<N; i++){
    a[i] = -i;
    b[i] = i*i;
  }

  cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
```

Vector addition: vectorsums.cu

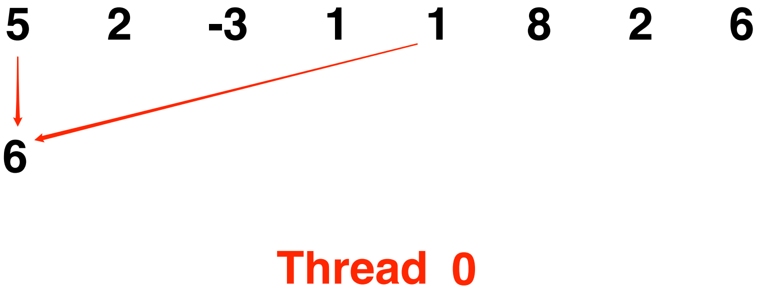
```

  add<<<N,1>>>(dev_a, dev_b, dev_c);

  cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

  printf("\na + b = c\n");
  for(i = 0; i<N; i++){
    printf("%5d + %5d = %5d\n", a[i], b[i], c[i]);
  }

  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);
}
```

Compiling and running vectorsums.cu

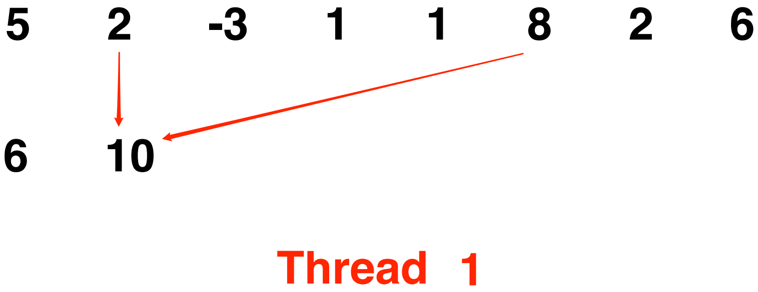
```
> nvcc vectorsums.cu -o vectorsums
> ./vectorsums
a + b = c
    0 +    0 =    0
   -1 +    1 =    0
   -2 +    4 =    2
   -3 +    9 =    6
   -4 +   16 =   12
   -5 +   25 =   20
   -6 +   36 =   30
   -7 +   49 =   42
   -8 +   64 =   56
   -9 +   81 =   72
```

Synchronizing threads within blocks: the pairwise sum revisited

- Example: pairwise sum of the vector (5, 2, -3, 1, 1, 8, 2, 6)



Synchronizing threads within blocks: the pairwise sum revisited



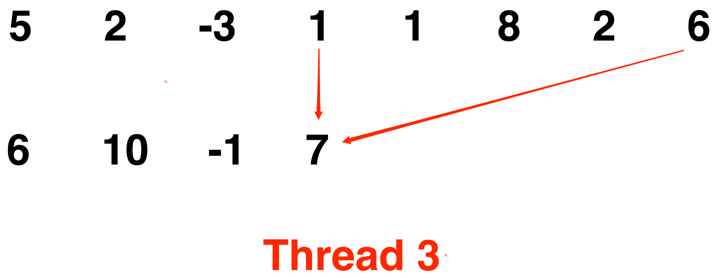
Synchronizing threads within blocks: the pairwise sum revisited

5 2 -3 1 1 8 2 6

6 10 -1

Thread 2

Synchronizing threads within blocks: the pairwise sum revisited



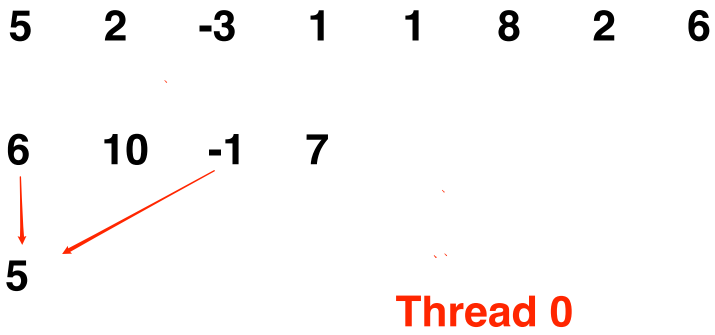
Synchronizing threads within blocks: the pairwise sum revisited

5 2 -3 1 1 8 2 6

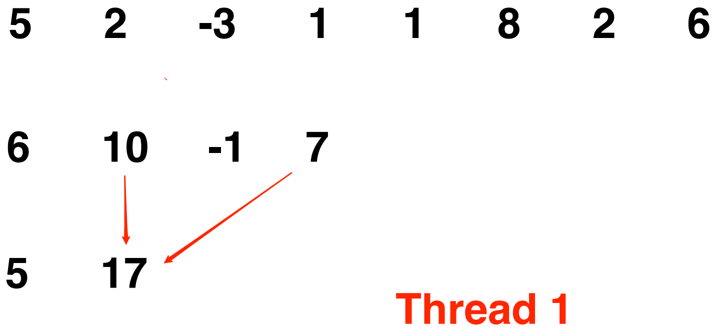
6 10 -1 7

Synchronize threads

Synchronizing threads within blocks: the pairwise sum revisited



Synchronizing threads within blocks: the pairwise sum revisited

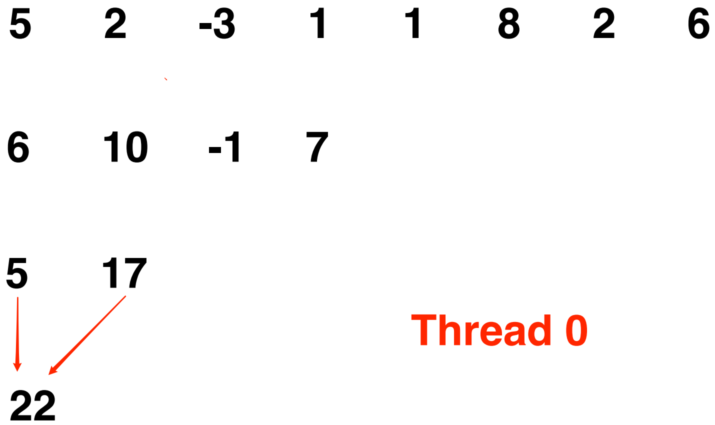


Synchronizing threads within blocks: the pairwise sum revisited

5	**2**	**-3**	**1**	**1**	**8**	**2**	**6**
6	**10**	**-1**	**7**				
5	**17**						

Synchronize Threads

Synchronizing threads within blocks: the pairwise sum revisited



Pairwise sum in pseudocode

- Let $n = 2^m$ be the length of the vector.
- Denote the vector by $(x_{(0,\ 0)}, \dots, x_{(0,\ n-1)})$
- Spawn 1 grid with a single block of $n/2$ threads.
- Do:
  1. Set offset $= n/2$.
  2. For parallel threads $j = 0, \dots, \text{offset} - 1$, compute:

     $$x_{(i,\ j)} = x_{(i-1,\ j)} + x_{(i-1,\ j+\text{offset})}$$

  3. Synchronize threads.
  4. Integer divide offset by 2.
  5. Return to step 2 if offset > 0.

pairwise_sum.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include <cuda_runtime.h>

/*
 * This program computes the sum of the elements of
 * vector v using the pairwise (cascading) sum algorithm.
 */

#define N 8 // length of vector v. MUST BE A POWER OF 2!!!

// Fill the vector v with n random floating point numbers.
void vfill(float* v, int n){
  int i;
  for(i = 0; i < n; i++){
    v[i] = (float) rand() / RAND_MAX;
  }
}

// Print the vector v.
void vprint(float* v, int n){
  int i;
  printf("v = \n");
  for(i = 0; i < n; i++){
    printf("%7.3f\n", v[i]);
  }
  printf("\n");
}
```

pairwise_sum.cu

```
// Pairwise-sum the elements of vector v and store the result in v
    [0].
__global__ void psum(float* v){
  int t = threadIdx.x; // Thread index.
  int n = blockDim.x; // Should be half the length of v.

  while (n != 0) {
    if(t < n)
      v[t] += v[t + n];
    __syncthreads();
    n /= 2;
  }
}

int main (void){
  float *v_h, *v_d; // host and device copies of our vector,
      respectively

  // dynamically allocate memory on the host for v_h
  v_h = (float*) malloc(N * sizeof(*v_h));

  // dynamically allocate memory on the device for v_d
  cudaMalloc ((float**) &v_d, N *sizeof(*v_d));

  // Fill v_h with N random floating point numbers.
  vfill(v_h, N);

  // Print v_h to the console
  vprint(v_h, N);
```

pairwise_sum.cu

```
  // Write the contents of v_h to v_d
  cudaMemcpy( v_d , v_h , N * sizeof(float), cudaMemcpyHostToDevice );

  // Compute the pairwise sum of the elements of v_d and store the
      result in v_d[0].
  psum<<< 1, N/2 >>>(v_d);

  // Write the pairwise sum, v_d[0], to v_h[0].
  cudaMemcpy(v_h, v_d, sizeof(float), cudaMemcpyDeviceToHost );

  // Print the pairwise sum.
  printf("Pairwise sum = %7.3f\n", v_h[0]);

  // Free dynamically-allocated host memory
  free(v_h);

  // Free dynamically-allocated device memory
  cudaFree(v_d);
}
```

Compiling and running pairwise_sum.cu

```
> nvcc pairwise_sum.cu -o pairwise_sum
> ./pairwise_sum
v =
  0.840
  0.394
  0.783
  0.798
  0.912
  0.198
  0.335
  0.768

Pairwise sum =    5.029
```

Best practices: respect the SIMD paradigm

- SIMD: "Single Instruction, Multiple Data"
- Under this paradigm, the thread in a kernel call write to different memory spaces.
- When threads write to the same memory (SISD), problems can arise.

sisd.cu: violating the SIMD paradigm

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void colonel(int *a_d){
  *a_d = blockDim.x * blockIdx.x + threadIdx.x;
}

int main(){

  int a = 0, *a_d;

  cudaMalloc((void**) &a_d, sizeof(int));
  cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

  colonel<<<4,5>>>(a_d);

  cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);

  printf("a = %d\n", a);
  cudaFree(a_d);

}
```

- What is the output?

sisd.cu: violating the SIMD paradigm

```
> nvcc sisd.cu -o sisd
> ./sisd
a = 14
```

- The output is unpredictable because the threads modify the same variable in an unpredictable order.

Outline

Resources

- Texts:
    1. J. Sanders and E. Kandrot. CUDA by Example. Addison-Wesley, 2010.
    2. D. Kirk, W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann, 2010.
- Code:
    - skeleton.cu
    - simple.cu
    - vectorsums.cu
    - pairwise_sum.cu

That's all for today.

- Series materials are available at http://will-landau.com/gpu.