INTRODUCTION TO PROGRAMMING IN CUDA C

Will Landau, Prof. Jarad Niemi

OUTLINE

- Defining and calling kernels and other device functions
- CPU-GPU communication
- Built-in CUDA C variables
- Synchronizing threads
- Respecting the SIMD paradigm

Featured examples:

- `skeleton.cu`
- `simple.cu`
- `vector_sums.cu`
- `pairwise_sum.cu`
- `sisd.cu`

BASIC C PROGRAM

```
#include <stdio.h>

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

BASIC CUDA C PROGRAM

```
#include <stdio.h>

__global__ void kernel(){
}

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

```
#include <stdio.h>

__global__ void kernel(){
}

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

Prefix that says, "only run this function on the GPU".

```
#include <stdio.h>

__global__ void kernel(){
}

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

CPU

Sent from CPU to GPU

CPU

CPU

__global__: Call from CPU and run only on GPU.

__device__: Call from GPU and run only on GPU.
(More specifically, call only from within
a __global__ or another __device__ function.)

__host__: Call from CPU and run only on CPU.
(i.e., a traditional C function.)

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

```
#include <stdio.h>

__device__ int dev1( void ){
}

__device__ int dev2( void ){
}

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

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

```
#include <stdio.h>

__global__ void kernel(){
}

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

?

GPU PARALLELISM

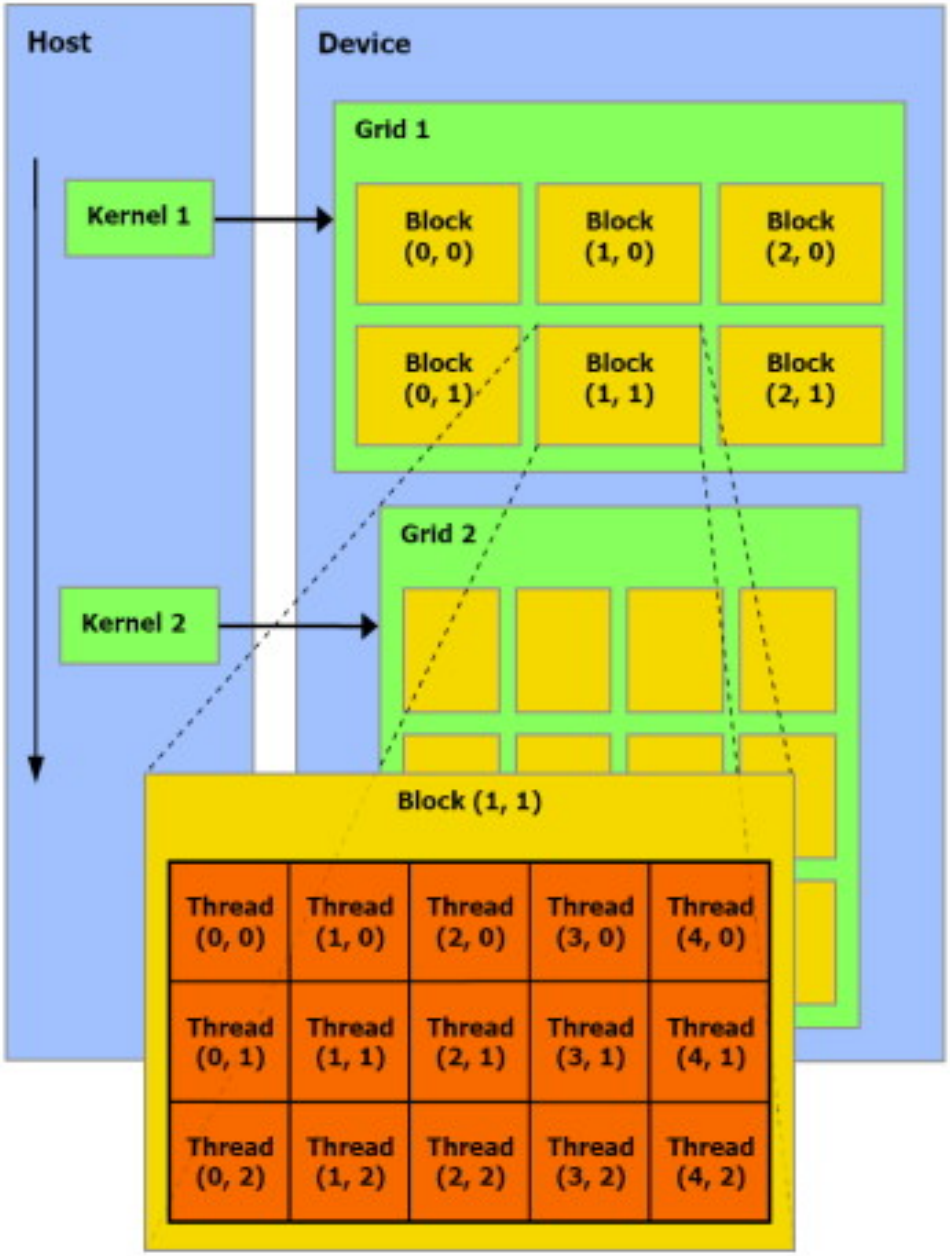
1. The CPU sends a CPU-to-GPU command called a **kernel** to a single GPU core.

2. The GPU core multitasks to execute the command:

   a. The GPU makes $B \cdot T$ copies of the kernel's code, and then runs all those copies simultaneously. Those parallel copies are called **threads**.

   b. The $B \cdot T$ threads are partitioned into B groups, called **blocks**, of T threads each.

   c. The sum total of all the threads from a kernel call is a **grid**.


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)

NOW BACK TO THE STUFF IN ANGLE BRACKTES...

```
#include <stdio.h>

__global__ void kernel(){
}

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

Here, the GPU runs kernel() one time:

```
#include <stdio.h>

__global__ void kernel ( void ) {
}

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

Here, the GPU runs kernel() 5 times:

```
#include <stdio.h>

__global__ void kernel ( void ) {
}

int main ( void ) {
  kernel<<<5,1>>>();
  printf( "Hello, World!" \n" )
  return 0;
}
```

Here, the GPU runs kernel() 5 times:

```
#include <stdio.h>

__global__ void kernel ( void ) {
}

int main ( void ) {
  kernel<<<1,5>>>();
  printf( "Hello, World!" \n" )
  return 0;
}
```

Here, the GPU runs kernel() 20 times:

```
#include <stdio.h>

__global__ void kernel ( void ) {
}

int main ( void ) {
  kernel<<<4,5>>>();
  printf( "Hello, World!" \n" )
  return 0;
}
```

BEYOND HELLO WORLD: skeleton.cu

```
#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 device memory back to host memory.
  ...
  // Free dynamically-allocated host memory
  ...
  // Free dynamically-allocated device memory
  ...
}
```

PASSING DATA TO AND FROM THE GPU: simple.cu

```
#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<<<100,100>>>(a_d);

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

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

```
}
```

```
[landau@impact1 simple]$ make
nvcc simple.cu -o simple
[landau@impact1 simple]$ ./simple
a = 2
[landau@impact1 simple]$
```

BUILT-IN CUDA C VARIABLES

- **maxThreadsPerBlock**: exactly that: 1024 on an impact1 core.

Also, within a kernel call with B blocks and T threads per block, you can use:

- **blockIdx.x**: the block ID corresponding to the current thread, an integer from 0 to $B - 1$ inclusive.

- **threadIdx.x**: the thread ID of the current thread within its block, an integer from 0 to $T - 1$ inclusive.

- **gridDim.x**: B, the number of blocks in the grid.

- **blockDim.x**: T, the number of threads per block.

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);

  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);
}
```

```
[landau@impact1 vectorsums]$ make
nvcc vectorsums.cu -o vectorsums
[landau@impact1 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
[landau@impact1 vectorsums]$
```

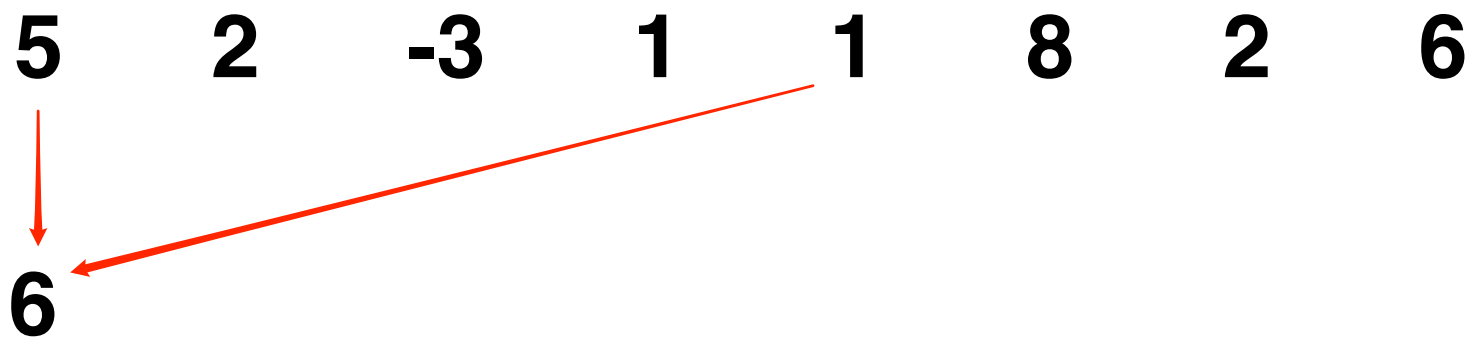
SYNCHRONIZING THREADS

Within a kernel call, whenever you need to synchronize all the threads in the current block, call:

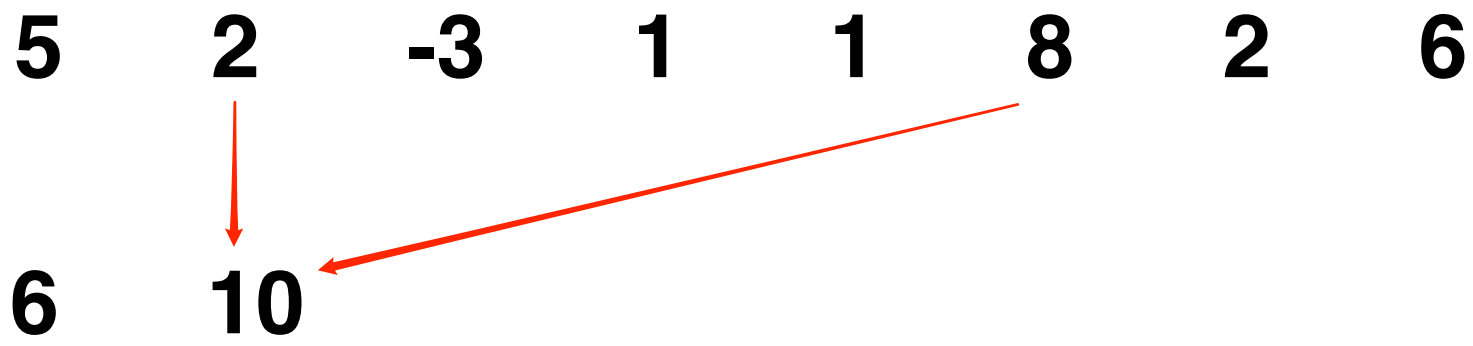
```
__syncthreads();
```

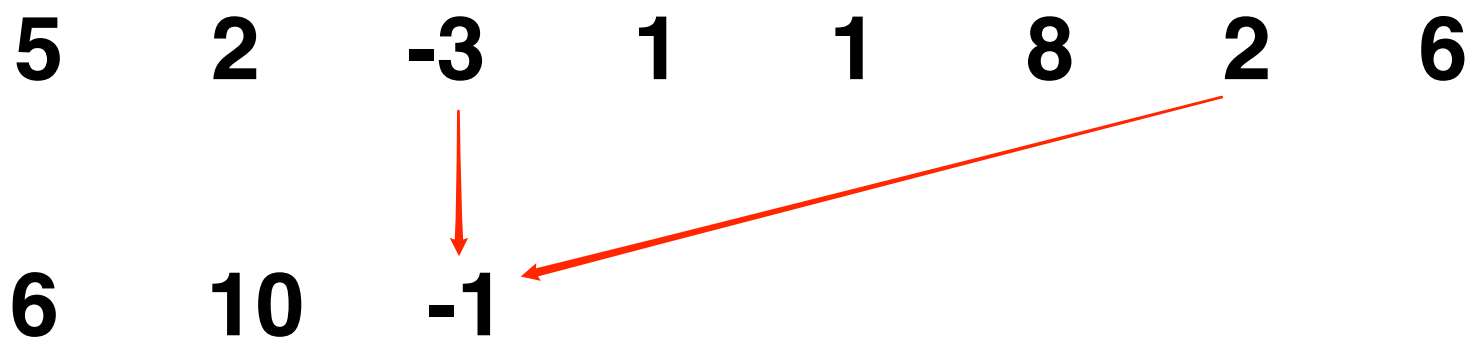
EXAMPLE: A RETURN TO THE PAIRWISE SUM

Let's take the pairwise sum of the vector, (5, 2, -3, 1, 1, 8, 2, 6):

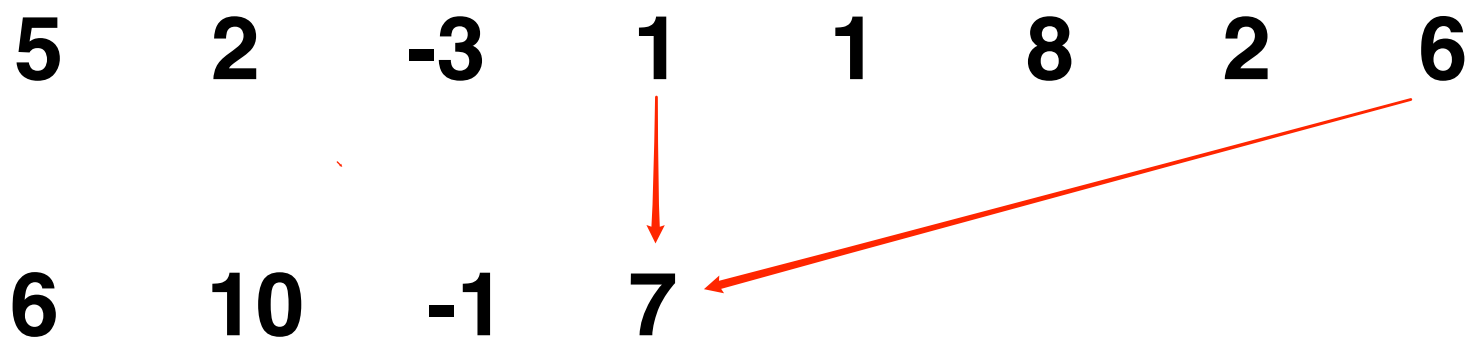


Thread 0


5 2 -3 1 1 8 2 6
6 10
Thread 1



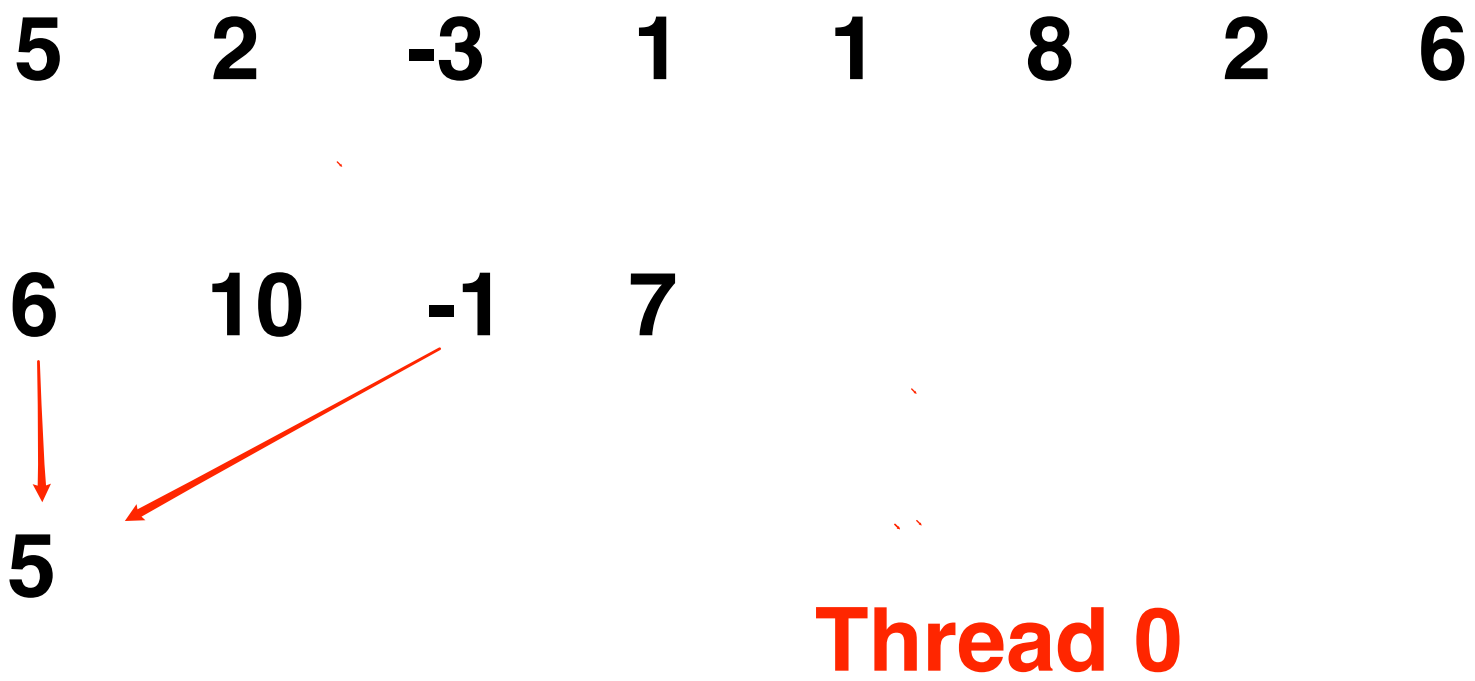
Thread 2

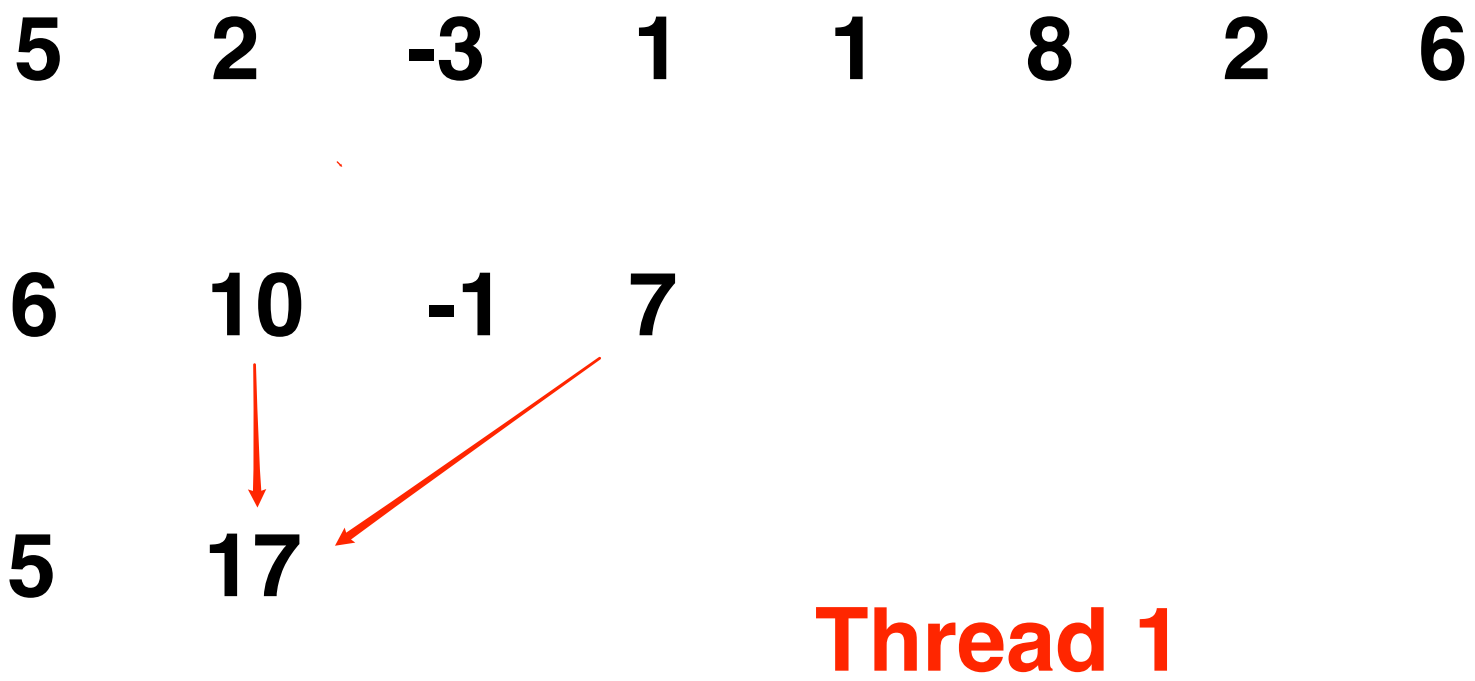

5 2 -3 1 1 8 2 6
6 10 -1 7
Thread 3

5 2 -3 1 1 8 2 6

6 10 -1 7

Synchronize threads


5 2 -3 1 1 8 2 6
6 10 -1 7
5
Thread 0


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

5 2 -3 1 1 8 2 6

6 10 -1 7

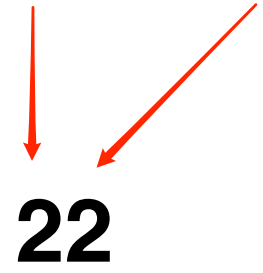
5 17

Synchronize Threads

5 2 -3 1 1 8 2 6

6 10 -1 7

5 17



22

Thread 0

A RIGOROUS DESCRIPTION OF THE PAIRWISE SUM

Suppose you have a vector $X_0 = (x_{(0,0)}, x_{(0,2)}, \ldots, x_{(0,n-1)})$, where $n = 2^m$ for some $m > 0$.

Compute $\sum_{i=1}^{n} x_{(0,i)}$ in the following way:

1. Create a new vector:
   $X_1 = (\underbrace{x_{(0,\ 0)} + x_{(0,\ n/2)}}_{x_{(1,\ 0)}},\ \underbrace{x_{(0,1)} + x_{(0,\ n/2+1)}}_{x_{(1,\ 1)}},\ \ldots, \underbrace{x_{(0,\ n/2-1)} + x_{(0,\ n-1)}}_{x_{(1,\ n/2-1)}})$
2. Create another new vector:
   $X_2 = (\underbrace{x_{(1,\ 0)} + x_{(1,\ n/4)}}_{x_{(2,\ 0)}},\ \underbrace{x_{(1,\ 1)} + x_{(1,\ n/4+1)}}_{x_{(2,\ 1)}},\ \ldots, \underbrace{x_{(1,n/4-1)} + x_{(1,\ n/2-1)}}_{x_{(2,\ n/4-1)}})$
3. Continue this process until you get a singleton vector:
   $X_m = (\underbrace{x_{(m-1,0)} + x_{(m-1,1)}}_{x_{(m,0)}})$

Notice: $\sum_{i=1}^{n} x_{(0,i)} = x_{(m,0)}$

PARALLELIZING THE PAIRWISE SUM

Spawn one grid with a single block and $n/2$ threads ($n = 2^m$). Starting with $i = 1$, do the following:

1. Set offset $= n/2^i$.
2. Assign thread j to compute:

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

   for $j = 0, 2, \cdots, \text{offset} - 1$.
3. Wait until all the above $\frac{n}{2^i}$ threads have completed step 2.
4. Integer divide offset by 2. Repeat if offset > 0.

PAIRWISE SUM: 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 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){
  if(N % 2){
    printf("\nERROR: N is not a power of 2. Exiting.\n");
    exit(1);
  }

  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);

  // 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);
}
```

THE SIMD PARADIGM

SIMD: Single Instruction Multiple Data

Each thread uses the same code, but applies it to different data.

Try to respect this paradigm in you code. If multiple threads access the same data, problems could arise.

EXAMPLE: `sisd.cu`

```
#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);

}
```

If we run this program, what will be the output?

```
[landau@impact1 sisd]$ make
nvcc sisd.cu -o sisd
[landau@impact1 sisd]$ ./sisd
a = 14
[landau@impact1 sisd]$
```

All the threads are trying to write to the same variable at the same time.

Hence, the value of `a` will correspond to the thread that finished last.

OUTLINE

- Defining and calling kernels and other device functions
- CPU-GPU communication
- Built-in CUDA C variables
- Synchronizing threads
- Respecting the SIMD paradigm

Featured examples:

- `skeleton.cu`
- `simple.cu`
- `vector_sums.cu`
- `pairwise_sum.cu`
- `sisd.cu`

LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

REFERENCES

David B. Kirk and Wen-mei W. Hwu. "Programming Massively Parallel Processors: a Hands-on Approach." Morgan Kaufman, 2010.

J. Sanders and E. Kandrot. *CUDA by Example.* Addison-Wesley, 2010.