CUDA – Beyond Basics

Piotr Luszczek

Mixing Blocks and Threads

```c
int N = 100, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  c[idx] = a[idx] + b[idx]; // no loop!
}
int main(void) {  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,10>>>(dev_a, dev_b, dev_c); // all 100 elements will be used

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf("%g\n", hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```
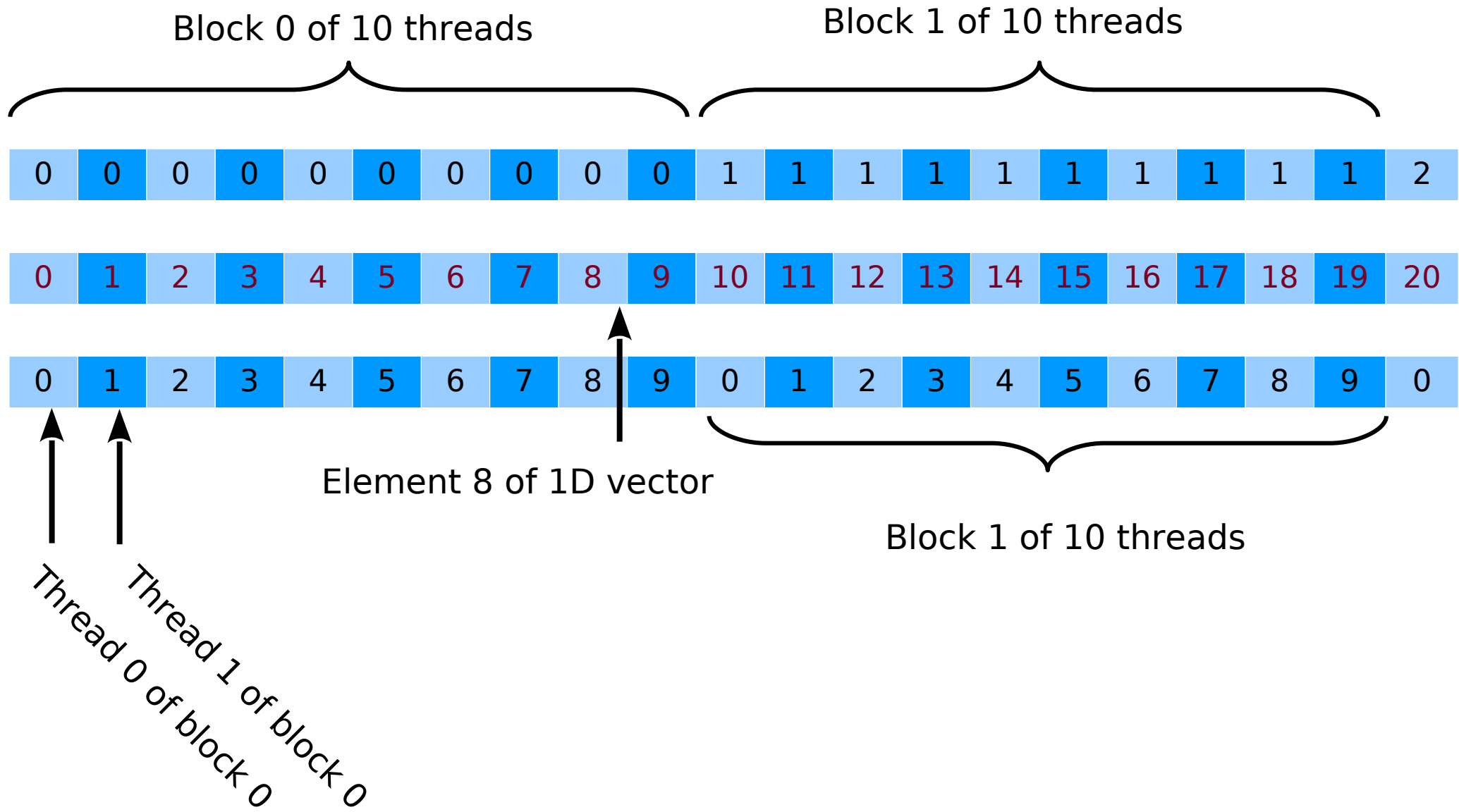
Block/Thread Indexing with <<<10,10>>>

Block 0 of 10 threads   Block 1 of 10 threads

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |

Element 8 of 1D vector

Block 1 of 10 threads

Thread 0 of block 0

Thread 1 of block 0

Mixing Blocks and Threads and Loop

```c
int N = 1000, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  int idx = (threadIdx.x + blockIdx.x * blockDim.x) * 10;
  for (int i=0; i<10; ++i) c[idx+i] = a[idx+i] + b[idx+i]; // use loop
}
int main(void) {  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,10>>>(dev_a, dev_b, dev_c);

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf("%g\n", hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```
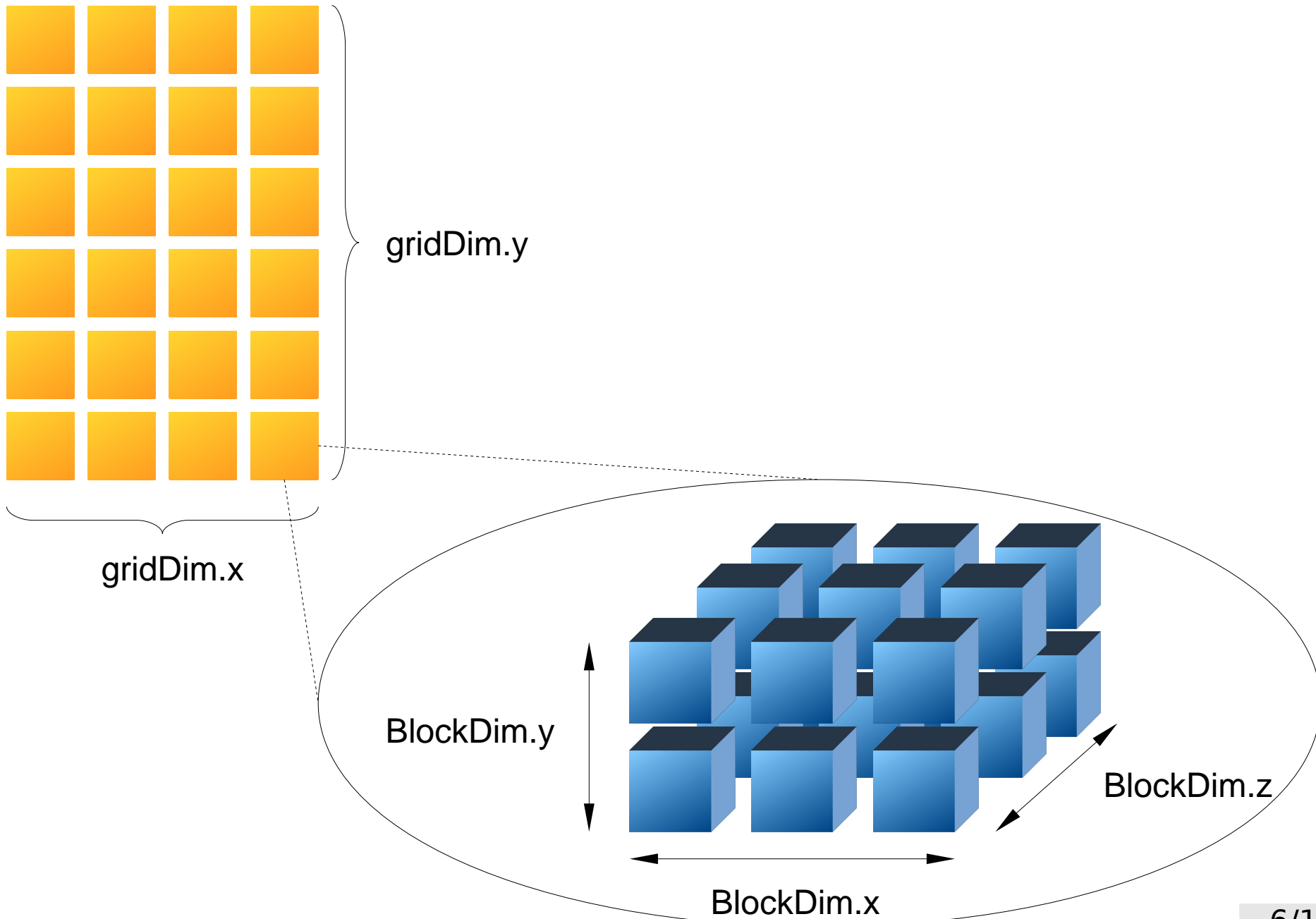
Grid, Blocks, Threads

- Complete syntax (almost):
  - kernel<<<gridDim, blockDim>>>
- CUDA API provides a data type: dim3
  - Grid of blocks:
    dim3 gridDim(grid_X_dimension, grid_Y_dimension)
  - Block of threads:
    dim3 blockDim(blk_X_d, blk_Y_d, blk_Z_d)

CUDA Grid of Blocks and Blocks of Threads

gridDim.y

gridDim.x

BlockDim.y

BlockDim.z

BlockDim.x

Limitations for and Early NVIDIA GPU

- GT200 was an early programmable GPU
- It had the following limits
  - 512 threads / block
  - 1024 threads / SM
  - 8 blocks / SM
  - 32 threads / warp

Comparison of Early NVIDIA GPUs

GPU	G80	GT200	GF100
Transistors (billions)	.681	1.4	3.0
CUDA Cores	128	240	512
Double precision FP	None	30 FMA ops / clock	256 FMA ops / clock
Single precision FP	128 MAD ops / clock	240 ops / clock	512 FMA ops / clock
Special Function Units / SM	2	2	4
Warp schedulers / SM	1	1	2
Shared memory / SM (KiB)	16	16	16 or 48
L1 Cache / SM (KiB)	None	None	16 or 48
L2 Cache (KiB)	None	None	768
ECC Memory Support	No	No	Yes
Concurrent kernels	No	No	1..16
Load/Store Address Bits	32	32	64

Calling Functions Inside Kernels

```
__device__ int fib(int n) { // don't do this!!!
  if (n > 2)
    return fib(n-2) + fib(n-1);
  return n;
}

__device__ int gpu_function(int ix) {
  return ix + 1;
}

__global__ void kernel(int *a, int *b, int *c) {
  c[0] = a[0] + gpu_function(b[0]);
}

int main(void) {
  kernel<<<10,10>>>(dev_a, dev_b, dev_c);

  return 0;
}
```

Synchronization Between Threads

- Threads within a block execute together and share:
  - L1 Cache
  - Shared memory
  - Split between the two is decided by kernel launch parameter
- Threads often do not execute all at the same time
  - But most of the time there are multiple threads executing
- They must synchronize
  - Synchronization is for all threads inside a single block
  - Blocks of threads are executed in arbitrary order
    - Gives CUDA runtime scheduling flexibility
- The most often used synchronization method
  - __syncthreads()

Sample Usage of Thread Synchronization

- Synchronization is invoked in kernel functions
  - `__global__ kernel() {`
    ```
    // parallel code section 0

    // wait for all threads to finish section 0
    __syncthreads();

    // parallel code section 1
    }
    ```
- Common mistake: not all threads reach synchronization
  - `__global__ error_kernel() {`
    ```
    if (threadIdx.x == 13) {
       // only some threads synchronize
       __syncthreads();
    } else
       // empty code path
    }
    ```

Asynchronous CUDA Calls

- **Recall the following**
  - The speed of the PCIexpress bus is slow
    - Use NVlink if your hardware supports it
  - Overlapping computation and communication
  - CPUs and GPUs are independent and work in parallel
  - There may be more than one CPU/GPU installed
- **There are asynchronous equivalents of many CUDA calls**
  - cudaMemcpy() has asynchronous equivalent cudaMemcpyAsync()
    - cudaMemcpy() will block CPU until the copy finishes
    - cudaMemcpyAsync() returns immediately and proceeds in the background
      - Keep in mind that CPU resources are needed to make progress
  - cudaDeviceSynchronize() blocks CPU until all past asynchronous calls complete