CUDA

More on threads, shared memory, synchronization

cuPrintf

- Library function for CUDA Developers
- Copy the files from /opt/cuPrintf into your source code folder

```
#include "cuPrintf.cu"

__global__ void testKernel(int val)
{
    cuPrintf("Value is: %d\n", val);
}
```

```
int main()
{
  cudaPrintfInit();

  testKernel<<< 2, 3 >>>(10);
  cudaPrintfDisplay(stdout, true);

  cudaPrintfEnd();
  return 0;
}
```

Handling Arbitrarily Long Vectors

- The limit is 512 threads per block, so there is a failure if the vector is of size N and N/512 > 65535
  - N > 65535*512 = 33,553,920 elements
  - Pretty big but we could have the capacity for up to 4GB
- Solution
  - Have to assign range of data values to each thread instead of each thread only operating on one value
- Next slide: An easy-to-code solution

Approach

- Have a fixed number of blocks and threads per block
  - Ideally some number to maximize the number of threads the GPU can handle per warp, e.g. 128 or 256 threads per block
- Each thread processes an element with a stride equal to the total number of threads.
- Example with 2 blocks, 3 threads per block, 10 element vector

Vector	0	1	2	3	4	5	6	7	8	9
Thread	B0T0	B0T1	B0T2	B1T0	B1T1	B1T2	B0T0	B0T1	B0T2	B1T0

Thread starts work at: (blockIdx.x * (NumBlocks)) + threadIdx.x
(blockIdx.x * blockDim.x) + threadIdx.x

e.g. B1T0 starts working at (1*2)+0 = index 3
next item to work on is at index 3 + TotalThreads = 3 + 6 = 9
↑
blockDim.x * gridDim.x

Vector Add Kernel For Arbitrarily Long Vectors

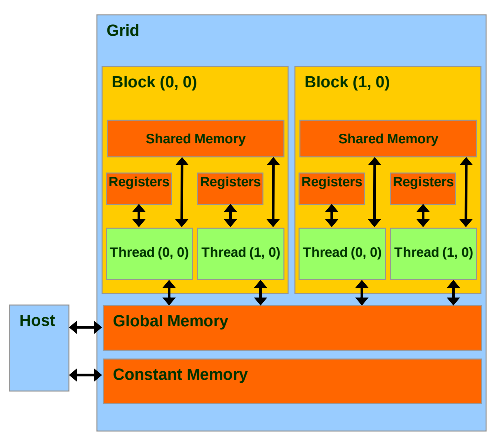
```
#define N (100 * 1024)     // Length of vector

__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;
  }
}
```

main: Pick some number of blocks less than N, threads to fill up a warp:

```
add<<<128, 128>>>(dev_a, dev_b, dev_c);        // 16384 total threads
```

G80 Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread **local memory**
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**
- Shared memory
  - Only shared among threads in the block
  - Is on chip, not DRAM, so fast to access
  - Useful for software-managed cache or scratchpad
  - Must be synchronized if the same value shared among threads



Shared Memory Example

- Dot Product
  - Book does a more complex version in matrix multiply
  - $(x_1x_2x_3x_4) \bullet (y_1y_2y_3y_4) = x_1y_1+x_2y_2+x_3y_3+x_4y_4$
  - When we did this with matrix multiply we had one thread perform this entire computation for a row and column
  - Obvious parallelism idea: Have thread_0 compute x_1y_1 and thread_1 compute x_2y_2, etc.
    - We have to store the individual products somewhere then add up all of the intermediate sums
    - Use shared memory

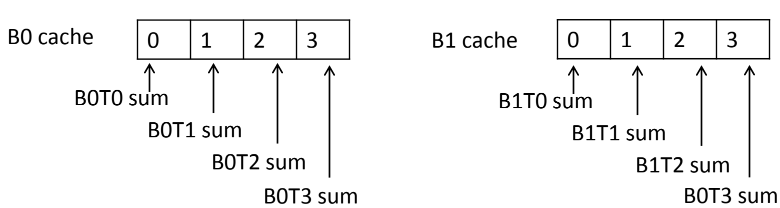
Shared Memory Dot Product

A	0	1	2	3	4	5	6	7	8	9
B	0	1	2	3	4	5	6	7	8	9
Thread	B0T0	B0T1	B0T2	B0T3	B1T0	B1T1	B1T2	B1T3	B0T0	B0T1

B0T0 computes A[0]*B[0] + A[8]*B[8]
B0T1 computes A[1]*B[1] + A[9]*B[9]
Etc. – this will be easier later with threadsPerBlock a power of 2

Store result in a per-block shared memory array:

```
__shared__ float cache[threadsPerBlock];
```



Kernel Test Code

```
#include "stdio.h"

#define N 10
const int THREADS_PER_BLOCK = 4;         // Have to be int, not #define; power of 2
const int NUM_BLOCKS = 2;                // Have to be int, not #define


__global__ void dot(float *a, float *b, float *c)
{
    __shared__ float cache[THREADS_PER_BLOCK];
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < N)
    {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;    // THREADS_PER_BLOCK * NUM_BLOCKS
    }
    cache[cacheIndex] = temp;
    if ((blockIdx.x == 0) && (threadIdx.x == 0))
        *c = cache[cacheIndex];           // For a test, only send back result of one thread
}
```

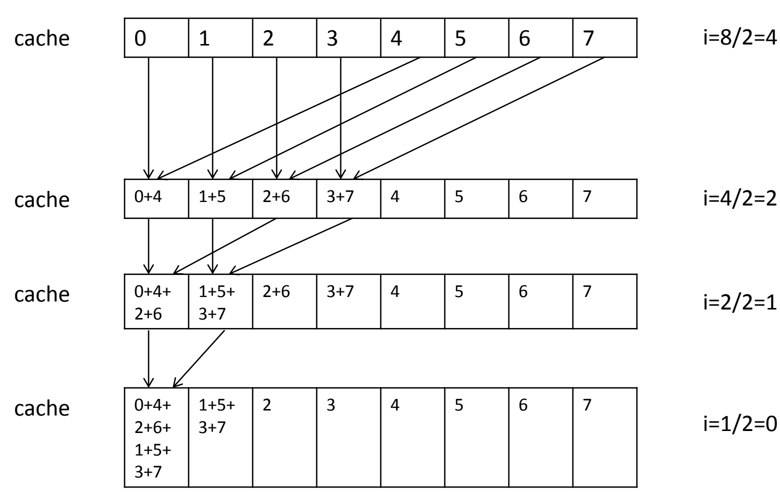
Main Test Code

```
int main()
{
    float a[N], b[N], c[NUM_BLOCKS];                // We'll see why c[NUM_BLOCKS] shortly
    float *dev_a, *dev_b, *dev_c;

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

    // Fill arrays
    for (int i = 0; i < N; i++)
    {
        a[i] = (float) i;
        b[i] = (float) i;
    }

    // Copy data from host to device
    cudaMemcpy(dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice);

    dot<<<NUM_BLOCKS,THREADS_PER_BLOCK>>>(dev_a,dev_b,dev_c);

    // Copy data from device to host
    cudaMemcpy(c, dev_c, NUM_BLOCKS*sizeof(float), cudaMemcpyDeviceToHost);

    // Output results
    printf("%f\n", c[0]);

    < cudaFree, return 0 would go here>
```

Accumulating Sums

- At this point we have products in cache[] in each block that we have to sum together
- Easy solution is to copy all these back to the host and let the host add them up
  - O(n) operation
  - If n is small this is the fastest way to go
- But we can do pairwise adds in logarithmic time
  - This is a common parallel algorithm called **reduction**

Summation Reduction



Have to wait for all working threads to finish adding before starting the next iteration

Summation Reduction Code

At the end of the kernel after storing temp into cache[cacheIndex]:

```
 int i = blockDim.x / 2;
while (i > 0)
{
        if (cacheIndex < i)
             cache[cacheIndex] += cache[cacheIndex + i];
         __syncthreads();
         i /= 2;
}
```

Summation Reduction Code

We still need to sum the values computed by each block. Since there are not too many of these (most likely) we just return the value to the host and let the host sequentially add them up:

```
 int i = blockDim.x / 2;
while (i > 0)
{
        if (cacheIndex < i)
             cache[cacheIndex] += cache[cacheIndex + i];
         __syncthreads();
         i /= 2;
}
```

```
if (cacheIndex == 0)                       // We're thread 0 in this block
    c[blockIdx.x] = cache[cacheIndex];     // Save the sum in array of blocks
```

Main

```
    dot<<<NUM_BLOCKS,THREADS_PER_BLOCK>>>(dev_a,dev_b,dev_c);

    // Copy data from device to host
    cudaMemcpy(c, dev_c, NUM_BLOCKS*sizeof(float), cudaMemcpyDeviceToHost);

    // Sum and output result
    float sum = 0;
    for (int i =0; i < NUM_BLOCKS; i++)
    {
        sum += c[i];
    }
    printf("The dot product is %f\n", sum);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return 0;
}
```

Thread Divergence

- When control flow differs among threads this is called thread divergence
- Under normal circumstances, divergent branches simply result in some threads remaining idle while others execute the instructions in the branch

```
int i = blockDim.x / 2;
while (i > 0)
{
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
}
```

Optimization Attempt

- In the reduction, only some of the threads (always less than half) are updating entries in the shared memory cache
- What if we only wait for the threads actually writing to shared memory

```
int i = blockDim.x / 2;
while (i > 0)
{
        if (cacheIndex < i)
        {
            cache[cacheIndex] += cache[cacheIndex + i];
            __syncthreads();
        }
        i /= 2;
}
```

Won't work; waits until ALL threads In the block reach this point

Summary

- There are some arithmetic details to map a block's thread to elements it should compute
- Shared memory is fast but only accessible by threads in the same block
- __syncthreads() is necessary when multiple threads access the same shared memory and must be used with care