The CUDA API wrap up

Memory Layout in CUDA

February 16, 2012

“If you don’t want to be replaced by a computer, don’t act like one.”

Arno Penzias
Before We Get Started…

- **Last time**
  - CMake, a tool for facilitating the software build process
  - Execution configuration in CUDA: grids, blocks, threads
  - Mapping a 3D thread index representation into a 1D thread id representation

- **Today**
  - API related issues
  - Simple matrix multiplication example
  - Memory allocation, copying, freeing, etc.

- **HW**
  - HW3: due today at 11:59 PM
  - HW4 emailed to you
Thread Index vs. Thread ID
[critical in understanding how SIMD is supported in CUDA & understanding the concept of “warp”]

- Each block organizes its threads in a 3D structure defined by its three dimensions: $D_x$, $D_y$, and $D_z$ that you specify.

- A block on Tesla C1060 cannot have more than 512 threads $\Rightarrow D_x \times D_y \times D_z \leq 512$.
  - Note: On Fermi architecture this is 1024.

- Each thread in a block can be identified by a unique index $(x, y, z)$, and

  $$0 \leq x \leq D_x \quad 0 \leq y \leq D_y \quad 0 \leq z \leq D_z$$

- A triplet $(x, y, z)$, called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as $t_{id} = x + y \times D_x + z \times D_x \times D_y$. You can regard this as a ”projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on ”warps” later).

- In general, operating for vectors typically results in you choosing $D_y = D_z = 0$. Handling matrices typically goes well with $D_z = 0$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.
Example: Adding Two Matrices

- You have two matrices A and B of dimension $N \times N$ ($N=32$)
- You want to compute $C = A + B$ in parallel
- Code provided below (some details omitted, such as `#define N 32`)

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of $N \times N \times 1$ threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Something to think about…

- Given that the device operates with groups of threads of consecutive ID, and given the scheme a few slides ago to compute a thread ID based on the thread & block index, is the array indexing scheme on the previous slide good or bad?

- The “good or bad” refers to how data is accessed in the device’s global memory.

- In other words should we have

\[ C[i][j] = A[i][j] + B[i][j] \]

or...

\[ C[j][i] = A[j][i] + B[j][i] \]
Combining Threads and Blocks

- Recall that there is a limit on the number of threads you can have in a block.

- In the vast majority of applications you need to use many blocks, each containing the same number of threads.

- Example: your assignment, when adding the two large vectors.
Indexing Arrays with Blocks and Threads

[important to grasp]

- No longer as simple as using only `threadIdx.x`
  - Consider indexing into an array, one thread accessing one element
  - Assume you have \( M=8 \) threads/block and the array has 32 entries

<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

- With \( M \) threads/block a unique index for each thread is given by:

\[
\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;
\]
Example: Indexing Arrays

What will be the array entry that thread of index 5 in block of index 2 will work on?

\[
\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;
\]
\[
= \quad 5 \quad + \quad 2 \quad \times \quad 8;
\]
\[
= \quad 21;
\]
A Recurring Theme in CUDA Programming
[and in SIMD in general]

- Imagine you are one of many threads, and you have your thread index and block index
  - You need to figure out what is the work you need to do
    - Just like on the previous slide, where thread 5 in block 2 had to deal with 21
  - You have to make sure you actually need to do that work
    - In many cases there are threads, typically of large id, that need to do no work
    - Example: you launch two blocks with 512 threads but your array is only 1000 elements long. Then 24 threads at the end do nothing
Vector Addition
[with Threads and Blocks: relevant in your assignment]

- Use the built-in variable `blockDim.x` for threads per block
  - This basically gives you the value of $M$ of two slides ago

```c
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- When it comes to launching the kernel, you’ll have to compute how many blocks you have to deal with:

```c
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

- How would you deal with a vector whose length $N$ is not a multiple of the number of threads $M$ in a block?

```c
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```
Timing Your Application
[useful for your assignment]

- Timing support – part of the CUDA API
  - You pick it up as soon as you include `<cuda.h>`

- Why is good to use
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

- Reports time in milliseconds with resolution of about 0.5 microseconds
  - From NVIDIA CUDA Library Documentation:
    - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns cudaErrorInvalidValue. If either event has been recorded with a non-zero stream, the result is undefined.
Timing Example
Timing a query of device 0 properties

```c++
#include <iostream>
#include <cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);
    cudaEventRecord(startEvent, 0);

    cudaDeviceProp deviceProp;
    const int currentDevice = 0;
    if (cudaGetDeviceProperties(&deviceProp, currentDevice) == cudaSuccess)
        printf("Device %d: %s\n", currentDevice, deviceProp.name);
    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";
    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```
Compiling CUDA

- Source files with CUDA language extensions must be compiled with `nvcc`
  - You spot such a file by its .cu suffix

- Example:
  
  ```
  >> nvcc -arch=sm_20 foo.cu
  ```

- Actually, `nvcc` is a compile driver
  - Works by invoking all the necessary tools and compilers like g++, cl, ...

- `nvcc` can output:
  - C code
    - Must then be compiled with the rest of the application using another tool
  - ptx code (CUDA’s ISA)
  - Or directly object code (cubin)
Compiling CUDA
[with nvcc driver]
PTX: Parallel Thread eXecution

- PTX: a pseudo-assembly language used in CUDA programming environment.
- `nvcc` translates code written in CUDA into PTX
- `nvcc` subsequently invokes a compiler which translates the PTX into a binary code which can be run on a certain GPU

```c
__global__ void fillKernel(int *a, int n)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        a[tid] = tid;
    }
}
```

PTX for `fillKernel`:
```ptx
.entry __Z10fillKernelPii ( .param .u64 __cudaparm__Z10fillKernelPii_a, .param .s32 __cudaparm__Z10fillKernelPii_n) { .reg .u16 %rh<4>; .reg .u32 %r<6>; .reg .u64 %rd<6>; .reg .pred %p<3>; .loc 14 5 0 $LDWbegin__Z10fillKernelPii: mov.u16 %rh1, %ctaid.x; mov.u16 %rh2, %ntid.x; mul.wide.u16 %r1, %rh1, %rh2; cvt.u32.u16 %r2, %tid.x; add.u32 %r3, %r2, %r1; ld.param.s32 %r4, [__cudaparm__Z10fillKernelPii_n]; setp.l.e.s32 %p1, %r4, %r3; @%p1 bra $Lt_0_1026; .loc 14 9 0 ld.param.u64 %rd1, [__cudaparm__Z10fillKernelPii_a]; cvt.s64.s32 %rd2, %r3; mul.wide.s32 %rd3, %r3, 4; add.u64 %rd4, %rd1, %rd3; st.global.s32 [%rd4+0], %r3; $Lt_0_1026: .loc 14 11 0 exit; $LDWend__Z10fillKernelPii: }
```
## The nvcc Compiler – Suffix Info

<table>
<thead>
<tr>
<th>File suffix</th>
<th>How the nvcc compiler interprets the file</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cu</td>
<td>CUDA source file, containing host and device code</td>
</tr>
<tr>
<td>.cup</td>
<td>Preprocessed CUDA source file, containing host code and device functions</td>
</tr>
<tr>
<td>.c</td>
<td>'C' source file</td>
</tr>
<tr>
<td>.cc, .cxx, .cpp</td>
<td>C++ source file</td>
</tr>
<tr>
<td>.gpu</td>
<td>GPU intermediate file (device code only)</td>
</tr>
<tr>
<td>.ptx</td>
<td>PTX intermediate assembly file (device code only)</td>
</tr>
<tr>
<td>.cubin</td>
<td>CUDA device only binary file</td>
</tr>
</tbody>
</table>

The CUDA API
What is an API?

- Application Programming Interface (API)
  - A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This is what replaced the old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program
On the CUDA API

- Reading the CUDA Programming Guide you’ll run into numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs `nvcc` compiler to generate an executable

- CUDA Driver API – low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly clunkier way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
CUDA API: Device Memory Allocation
[Note: picture assumes two blocks, each with two threads]

- **cudaMalloc()**
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- **cudaFree()**
  - Frees object from device Global Memory
    - Pointer to freed object
Example Use: A Matrix Data Type

- NOT part of CUDA API
- Used in several code examples
  - 2 D matrix
  - Single precision float elements
  - \( \text{width} \times \text{height} \) entries
  - Matrix entries attached to the pointer-to-float member called "elements"
  - Matrix is stored row-wise

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Example
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to \texttt{Md.elements}
  - \texttt{“d”} in “Md” is often used to indicate a device data structure

```c
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);

//use it for what you need, then free the device memory
cudaFree(Md.elements);
```
CUDA Host-Device Data Transfer

- **cudaMemcpy()**
  - Memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
  - Type of transfer
    - Host to Host
    - Host to Device
    - Device to Host
    - Device to Device
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```
Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs

- Use only global memory (don’t bring shared memory into picture yet)

- Concentrate on
  - Thread ID usage

- Memory data transfer API between host and device
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $\text{WIDTH} \times \text{WIDTH}$

- Software Design Decisions:
  - One thread handles one element of $P$
  - Each thread will access all the entries in one row of $M$ and one column of $N$
    - $2 \times \text{WIDTH}$ read accesses to global memory
    - One write access to global memory
Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P

- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…

- Size of matrix limited by the number of threads allowed in a thread block
/ Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; //march along a row of M
                double b = N.elements[k * N.width + j]; //march along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
Step 2: Matrix Multiplication

[host-side code]

```c
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P) {
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 4: Matrix Multiplication - Device-side Kernel Function

```c
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]...
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P. width + tx] = Pvalue
    float Pvalue = 0;

    for (int k = 0; k < M.width; ++k) {
        float Melement = M.elements[ty * M.width + k];
        float Nelement = N.elements[k * N.width + tx];
        Pvalue += Melement * Nelement;
    }

    // Write matrix to device memory; each thread one element
    P.elements[ty * P. width + tx] = Pvalue;
}
```
Step 4: Some Loose Ends

```c
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMemcpy((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```
CUDA runtime API: exposes a set of extensions to the C language
- See Section 4.1 and Appendix B of “NVIDIA CUDA C Programming Guide”
  - Keep in mind the 20/80 rule

It consists of:
- Language extensions
  - To target portions of the code for execution on the device

- A runtime library, which is split into:
  - A common component providing built-in vector types and a subset of the C runtime library available in both host and device codes
    - Callable both from device and host
  - A host component to control and access devices from the host
    - Callable from the host only
  - A device component providing device-specific functions
    - Callable from the device only
**Language Extensions: Variable Type Qualifiers**

<table>
<thead>
<tr>
<th><strong>device</strong> <strong>local</strong></th>
<th>int LocalVar;</th>
<th>Memory: local</th>
<th>Scope: thread</th>
<th>Lifetime: thread</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>int SharedVar;</td>
<td>Memory: shared</td>
<td>Scope: block</td>
<td>Lifetime: block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>int GlobalVar;</td>
<td>Memory: global</td>
<td>Scope: grid</td>
<td>Lifetime: application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>int ConstantVar;</td>
<td>Memory: constant</td>
<td>Scope: grid</td>
<td>Lifetime: application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__

- **Automatic variables** without any qualifier reside in a register
  - Except arrays, which reside in local memory (unless they are small and of known constant size)
Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host.

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes
Common Runtime Component: Built-in Vector Types

- [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double[1..2]
  - Structures accessed with \( x, y, z, w \) fields:
    ```
    uint4 param;
    int dummy = param.y;
    ```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in “NVIDIA CUDA C Programming Guide”
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asinh`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- etc.

  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions only supported for scalar types, not vector types
Host Runtime Component

- Provides functions available only to the host to deal with:
  - **Device** management (including multi-device systems)
  - **Memory** management
  - **Error** handling

- **Examples**
  - **Device memory allocation**
    - cudaMalloc(), cudaFree()
  - **Memory copy from host to device, device to host, device to device**
    - cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()
  - **Memory addressing** – returns the address of a device variable
    - cudaGetSymbolAddress()
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. \( \sin(x) \)) have a less accurate, but faster device-only version (e.g. \( \_\_\sin(x) \))
  - \( \_\_\text{pow} \)
  - \( \_\_\text{log}, \_\_\text{log2}, \_\_\text{log10} \)
  - \( \_\_\text{exp} \)
  - \( \_\_\sin, \_\_\cos, \_\_\tan \)
- Some of these have hardware implementations
- By using the “-use_fast_math” flag, \( \sin(x) \) is substituted at compile time by \( \_\_\sin(x) \)

\[
\text{>> nvcc} \quad \text{-arch=sm\_20} \quad \text{-use\_fast\_math} \quad \text{foo.cu}
\]