INTRODUCTION TO CUDA C++

Jeff Larkin, June 28, 2018
CUDA C/C++ AND FORTRAN

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
HETEROGENEOUS COMPUTING

Terminology:

*Host*  
The CPU and its memory (host memory)

*Device*  
The GPU and its memory (device memory)
SIMPLE EXECUTION MODEL
NVCC COMPILER

NVIDIA provides a CUDA-C compiler

nvcc

NVCC splits your code in 2: Host code and Device code.

Host code forwarded to CPU compiler (usually g++)

Device code sent to NVIDIA device compiler

NVCC is capable of linking together both host and device code into a single executable

Convention: C++ source files containing CUDA syntax are typically given the extension .cu.
EXAMPLE 1: HELLO WORLD

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

Terminology:

“Kernel” - A function called on the GPU by all threads participating in a calculation.
OUR FIRST KERNEL

```c
__global__ void mykernel(void) {
}
```

The `__global__` annotation informs the compiler that this is a kernel, which will be invoked on the device from the host.

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

The angle bracket, or “chevron”, syntax informs the compiler how many copies of the kernel “mykernel” to invoke. Here we will invoke is once.
OUR FIRST KERNEL

```c
__global__ void mykernel(void) {
    printf("Hello, World!\n");
}

int main(void) {
    mykernel<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Move the work into the kernel.

Tell the host to wait until the device is finished.
COMPILING AND RUNNING

Compile the code with NVCC

$ nvcc main.cu

Run the resulting executable

$ ./a.out

Hello, World!
PARALLEL PROGRAMMING IN CUDA C/C++

But wait... GPU computing is about massive parallelism!

We need a more interesting example...

We’ll start by adding two integers and build up to vector addition
EXAMPLE 2: VECTOR ADDITION

void vecadd(int *a, int *b, int *c, int N) {
    for(int i=0; i<N; i++)
        c[i] = a[i] + b[i];
}

Plan of Attack:

1. Move addition to element-wise function
2. Make new function a kernel
3. Make vectors available on the device
4. Invoke the new GPU kernel
VECADD: STEP 1, ELEMENT-WISE FUNCTION

// Compute 1 element of c from a and b
void vecadd_kernel(int *a, int *b, int *c, int N, int i) {
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    for(int i=0;i<N;i++)
        vecadd_kernel(a, b, c, N, i);
}
THREAD HIERARCHY IN CUDA

Thread

Thread Block

Grid
// Compute 1 element of c from a and b
__global__ void vecadd_kernel(int *a, int *b, int *c, int N) {
    int i = threadIdx.x; // Calculate my index
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    for(int i=0; i<N; i++)
        vecadd_kernel(a, b, c, N, i);
}
int main() {
    int N=512;
    int *a, *b, *c;

    a=(int*)malloc(N*sizeof(int));
    b=(int*)malloc(N*sizeof(int));
    c=(int*)malloc(N*sizeof(int));
    ...
    vecadd(a, b, c, N);
    ...

    free(a);
    free(b);
    free(c);
    return 0;
}
CUDA Memory Management

No Unified Memory

Unified Memory

System Memory

GPU Memory

Unified Memory
int main() {
    int N=512;
    int *a, *b, *c;

    cudaMallocManaged(&a, N*sizeof(int));
    cudaMallocManaged(&b, N*sizeof(int));
    cudaMallocManaged(&c, N*sizeof(int));

    ...
    vecadd(a, b, c, N);
    ...

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
VECADD: STEP 4, INVOKE KERNEL

// Compute 1 element of c from a and b
__global__ void vecadd_kernel(int *a, int *b, int *c, int N) {
    int i = threadIdx.x; // Calculate my index
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    vecadd_kernel<<<1,N>>>(a, b, c, N);
    cudaDeviceSynchronize();
}

Launch vecadd_kernel() on 1 thread block with N threads.

Ensure kernel completes before vecadd() returns.
// Compute 1 element of c from a and b
__global__ void vecadd_kernel(int *a, int *b, int *c, int N) {
    int i = threadIdx.x; // Calculate my index
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    vecadd_kernel<<<N,1>>>(a, b, c, N);
    cudaDeviceSynchronize();
}

Launch vecadd_kernel() on N thread block with 1 thread.

Ensure kernel completes before vecadd() returns.
COMBINING BLOCKS AND THREADS

We’ve seen parallel vector addition using:
- Several blocks with one thread each
- One block with several threads

To utilize all the cores we need to use both blocks and threads

Let’s adapt vector addition to use both blocks and threads

First let’s discuss data indexing...
BUILT-IN VARIABLES

Built-in variables:

- threadIdx.x: Thread index within the block
- blockIdx.x: Block index within the grid
- blockDim.x: Number of threads in a block
- gridDim.x: Number of blocks in a grid

These exist automatically in CUDA kernels
Read only (set by the runtime)
INDEXING ARRAYS WITH BLOCKS AND THREADS

No longer as simple as using `blockIdx.x` and `threadIdx.x`

Consider indexing an array with one element per thread (8 threads/block)

With `blockDim.x` threads per block, a unique index for each thread is given by:

```
int index = blockIdx.x * blockDim.x + threadIdx.x
```
INDEXING ARRAYS: EXAMPLE

Which thread will operate on the red element?

```
int index = blockIdx.x * blockDim.x + threadIdx.x
= 2 * 8 + 5;
= 21;
```
VECADD: STEP 4, INVOKE KERNEL

// Compute 1 element of c from a and b
__global__ void vecadd_kernel(int *a, int *b, int *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    vecadd_kernel<<<N/1024,1024>>>(a, b, c, N);
    cudaDeviceSynchronize();
}

Launch vecadd_kernel() on N/1024 thread blocks of 1024 threads.

Ensure kernel completes before vecadd() returns.
// Compute 1 element of c from a and b
__global__ void vecadd_kernel(int *a, int *b, int *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) // Protect against out-of-bounds error
        c[i] = a[i] + b[i];
}

void vecadd(int *a, int *b, int *c, int N) {
    vecadd_kernel<<<(N+1023)/1024/1024,1024>>>(a, b, c, N);
    cudaDeviceSynchronize();
}

If N is not evenly divisible by 1024, this will ensure enough blocks are created to cover all data elements.
CUDA MEMORY MANAGEMENT

Without Unified Memory

```c
void sortfile(FILE *fp, int N) {
    char *data, *d_data;
    data = (char*) malloc(N);
    cudaMemcpy (&d_data, N);

    fread(data, 1, N, fp);
    cudaMemcpy (d_data, data, N, H2D);
    qsort <<<...>>> (d_data, N, 1, compare);
    cudaMemcpy (data, d_data, N, D2H);

    use_data (data);
    free(data);
    cudaFree (d_data);
}
```

Unified Memory

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged (&data, N);

    fread(data, 1, N, fp);
    qsort <<<...>>> (data, N, 1, compare);
    cudaMemcpy (data, d_data, N, D2H);

    use_data (data);
    cudaFree (data);
}
```

CUDA MEMORY MANAGEMENT

cudaMalloc & cudaMemcpy
- Explicitly track host and device memory
- Explicitly relocate data (sync or async)
- Expresses data locality (most performance)

cudaMallocManaged
- Single pointer for host & device memory
- Transfer at launch and sync
- Data paged to the host on demand
- Device paging from the host in future hardware

Advice: Develop with cudaMallocManaged then optimize to cudaMalloc/cudaMemcpy if necessary
int main() {
    int N=512;
    int *a, *a_d, *b, *b_d, *c, *c_d;
    cudaMallocHost(&a,N*sizeof(int));
    cudaMallocHost(&b,N*sizeof(int));
    cudaMallocHost(&c,N*sizeof(int));
    cudaMalloc(&a_d,N*sizeof(int));
    cudaMalloc(&b_d,N*sizeof(int));
    cudaMalloc (&c_d,N*sizeof(int));
    ...;
    cudaMemcpy(a_d, a, N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b, N*sizeof(int),cudaMemcpyHostToDevice);
    vecadd(a_d, b_d, c_d, N);
    cudaMemcpy(c, c_d, N*sizeof(int),cudaMemcpyDeviceToHost);
    ...;
}
CLOSING SUMMARY

CUDA C/C++ and Fortran provide close-to-the-metal performance, but may require rethinking your code.

CUDA programming explicitly replaces loops with parallel kernel execution.

Using CUDA Managed Memory simplifies data management by allowing the CPU and GPU to dereference the same pointer.