Introduction to GPU Programming

Volodymyr (Vlad) Kindratenko
Innovative Systems Laboratory @ NCSA
Institute for Advanced Computing Applications and Technologies (IACAT)
Part II

• CUDA C

• Hands-on: Mandelbrot set fractal renderer
  – Reference implementation
  – GPU implementation

• Hands-on: reduction kernel
  – Reference implementation
  – GPU port

• Lunch
CUDA C

• CUDA C extends standard C as follows
  – Function type qualifiers to specify whether a function executes on the host or on the device
  – Variable type qualifiers to specify the memory location on the device
  – A new directive to specify how a kernel is executed on the device
  – Four built-in variables that specify the grid and block dimensions and the block and thread indices
  – Built-in vector types derived from basic integer and float types
Vector types derived from basic integer and float types

- char1, char2, char3, char4
- uchar1, uchar2, uchar3, uchar4
- short1, short2, short3, short4
- ushort1, ushort2, ushort3, ushort4
- int1, int2, int3, int4
- uint1, uint2, uint3 (dim3), uint4
- long1, long2, long3, long4
- ulong1, ulong2, ulong3, ulong4
- longlong1, longlong2
- float1, float2, float3, float4
- double1, double2

They are all structures, like this:

```c
typedef struct {
  float x, y, z, w;
} float4;
```

They all come with a constructor function in the form `make_<typeName>`, e.g.,

```c
int2 make_int2(int x, int y);
```
Example

• dim3 dimBlock(width, height);
• dim3 dimGrid(10);  // same as dimGrid(10,0,0)

• myKernel<<dimGrid, dimBlock>>();
## Built-in Variables

<table>
<thead>
<tr>
<th>variable</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>dim3</td>
<td>dimensions of the grid</td>
</tr>
<tr>
<td>blockID</td>
<td>unit3</td>
<td>block index within the grid</td>
</tr>
<tr>
<td>blockDim</td>
<td>dim3</td>
<td>dimensions of the block</td>
</tr>
<tr>
<td>threadIdx</td>
<td>uint3</td>
<td>thread index within the block</td>
</tr>
<tr>
<td>warpSize</td>
<td>int</td>
<td>warp size in threads</td>
</tr>
</tbody>
</table>

It is not allowed to take addresses of any of the built-in variables.
It is not allowed to assign values to any of the built-in variables.
Example

myKernel<<10, 32>>();

__global__ void myKernel()
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

• here
  – gridSize.x is 10
  – blockDim.x is 32
## Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable Type Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__</code> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><code>__device__ __shared__</code> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><code>__device__ __constant__</code> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><code>volatile</code> int GlobarVar or SharedVar;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `__shared__` and `__constant__` variables have implied static storage
- `__device__`, `__shared__` and `__constant__` variables cannot be defined using external keyword
- `__device__` and `__constant__` variables are only allowed at file scope
- `__constant__` variables cannot be assigned to from the devices, they are initialized from the host only
- `__shared__` variables cannot have an initialization as part of their declaration
Example

```c
__global__ void myKernel()
{
    __shared__ float shared[32];
    __device__ float device[32];
    shared[threadIdx.x] = device[threadIdx.x];
}
```
Example

```c
__global__ void myKernel()
{
    extern __shared__ int s_data[];

    s_data[threadIdx.x] = ...
}

main()
{
    int sharedMemSize = numThreadsPerBlock * sizeof(int);
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    myKernel <<< dimGrid, dimBlock, sharedMemSize >>>();
}
```
Function Type Qualifiers

<table>
<thead>
<tr>
<th><strong>device</strong> float DeviceFunc()</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

__device__ and __global__ functions do not support recursion, cannot declare static variables inside their body, cannot have a variable number of arguments __device__ functions cannot have their address taken __host__ and __device__ qualifiers can be used together, in which case the function is compiled for both __global__ and __host__ qualifiers cannot be used together __global__ function must have void return type, its execution configuration must be specified, and the call is asynchronous
Example

__device__ int get_global_index(void)
{
    return blockIdx.x * blockDim.x + threadIdx.x;
}

__global__ void myKernel(int *array)
{
    int index = get_global_index();
}

main()
{
    ... 
    myKernel<<<gridSize, blockSize>>>(gArray);
    ... 
}
Function declared as

```
__global__ void kernel(float* param);
```

must be called like this:

```
kernel<<<Dg, Db, Ns, S>>>(param);
```

where

- **Dg** (type dim3) specifies the dimension and size of the grid, such that \(Dg.x \times Dg.y\) equals the number of blocks being launched;
- **Db** (type dim3) specifies the dimension and size of each block of threads, such that \(Db.x \times Db.y \times Db.z\) equals the number of threads per block;
- optional **Ns** (type size_t) specifies the number of bytes of shared memory dynamically allocated per block for this call in addition to the statically allocated memory
- optional **S** (type cudaStream_t) specifies the stream associated with this kernel call
Intrinsic Functions

Supported on the device only

Start with __, as in __sinf(x)

End with
__rn (round-to-nearest-even rounding mode)
__rz (round-towards-zero rounding mode)
__ru (round-up rounding mode)
__rd (round-down rounding mode)
as in __fadd_rn(x,y);

There are mathematical (__log10f(x)), type conversion (__int2float_rn(x)), type casting (__int_as_float(x)), and bit manipulation (__ffs(x)) functions
Example

```cpp
__global__ void myKernel(float *a1, float *a2)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    a1[index] = sin(a1[index]);

    // faster, but less precise than sin()
    a2[index] = __sin_rn(a2[index]);
}
```
## Synchronization and Memory Fencing Functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void __threadfence()</code></td>
<td>wait until all global and shared memory accesses made by the calling thread become visible to all threads in the device for global memory accesses and all threads in the thread block for shared memory accesses</td>
</tr>
<tr>
<td><code>void __threadfence_block()</code></td>
<td>Waits until all global and shared memory accesses made by the calling thread become visible to all threads in the thread block</td>
</tr>
<tr>
<td><code>void __syncthreads()</code></td>
<td>Waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads become visible to all threads in the block</td>
</tr>
</tbody>
</table>
Example

```c
__global__ void myKernel(float *a1, float *a2)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    a1[index] = a1[index] + a2[index];

    __syncthreads();

    a2[index] = a1[blockDim.x-index-1];
}
```
## Atomic Functions

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomicAdd()</td>
<td>new = old + val</td>
</tr>
<tr>
<td>atomicSub()</td>
<td>new = old − val</td>
</tr>
<tr>
<td>atomicExch()</td>
<td>new = val</td>
</tr>
<tr>
<td>atomicMin()</td>
<td>new = min(old, val)</td>
</tr>
<tr>
<td>atomicMax()</td>
<td>new = max(old, val)</td>
</tr>
<tr>
<td>atomicInc()</td>
<td>new = ((old &gt;= val) ? 0 : (old+1))</td>
</tr>
<tr>
<td>atomicDec()</td>
<td>new = (((old==0)</td>
</tr>
<tr>
<td>atomicCAS()</td>
<td>new = (old == compare ? val : old)</td>
</tr>
<tr>
<td>Atomic{And, Or, Xor}()</td>
<td>new = {(old &amp; val), (old</td>
</tr>
</tbody>
</table>

An atomic function performs read-modify-write atomic operation on one 32-bit or one 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.
__shared__ totalSum;
if (threadIdx.x == 0) totalSum = 0;
__syncthreads();

int localVal = pValues[blockIdx.x * blockDim.x + threadIdx.x];
atomicAdd(&totalSum, 1);
__syncthreads();
## Device Management

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaGetDeviceCount()</td>
<td>Returns the number of compute-capable devices</td>
</tr>
<tr>
<td>cudaGetDeviceProperties()</td>
<td>Returns information on the compute device</td>
</tr>
<tr>
<td>cudaSetDevice()</td>
<td>Sets device to be used for GPU execution</td>
</tr>
<tr>
<td>cudaGetDevice()</td>
<td>Returns the device currently being used</td>
</tr>
<tr>
<td>cudaChooseDevice()</td>
<td>Selects device that best matches given criteria</td>
</tr>
</tbody>
</table>
Device Management Example

```c
void cudaDeviceInit() {
    int devCount, device;
    cudaGetDeviceCount(&devCount);
    if (devCount == 0) {
        printf("No CUDA capable devices detected.\n");
        exit(EXIT_FAILURE);
    }
    for (device=0; device < devCount; device++) {
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, device);
        // If a device of compute capability >= 1.3 is found, use it
        if (props.major > 1 || (props.major == 1 && props.minor >= 3)) break;
    }
    if (device == devCount) {
        printf("No device above 1.2 compute capability detected.\n");
        exit(EXIT_FAILURE);
    }
    else cudaSetDevice(device);
}
```
# Memory Management

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMalloc()</td>
<td>Allocates memory on the GPU</td>
</tr>
<tr>
<td>cudaMallocPitch()</td>
<td>Allocates memory on the GPU device for 2D arrays, may pad the allocated memory to ensure alignment requirements</td>
</tr>
<tr>
<td>cudaFree()</td>
<td>Frees the memory allocated on the GPU</td>
</tr>
<tr>
<td>cudaMallocArray()</td>
<td>Allocates an array on the GPU</td>
</tr>
<tr>
<td>cudaFreeArray()</td>
<td>Frees an array allocated on the GPU</td>
</tr>
<tr>
<td>cudaMallocHost()</td>
<td>Allocates page-locked memory on the host</td>
</tr>
<tr>
<td>cudaFreeHost()</td>
<td>Frees page-locked memory in the host</td>
</tr>
</tbody>
</table>
## Memory Management (Cont.)

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMemset()</td>
<td>Initializes or sets GPU memory to a value</td>
</tr>
<tr>
<td>cudaMemCpy()</td>
<td>Copies data between host and the device</td>
</tr>
<tr>
<td>cudaMemcpyToArray()</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyFromArray()</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyArrayToArray()</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyToSymbol()</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyFromSymbol()</td>
<td></td>
</tr>
<tr>
<td>cudaGetSymbolAddress()</td>
<td>Finds the address associated with a CUDA symbol</td>
</tr>
<tr>
<td>cudaGetSymbolSize()</td>
<td>Finds the size of the object associated with a CUDA symbol</td>
</tr>
</tbody>
</table>
Example

main()
{
...

float *devPtrA, *devPtrB;

cudaMalloc((void**)&devPtrA, N * sizeof(float));
cudaMemcpy(devPtrA, A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMalloc((void**)&devPtrB, N * sizeof(float));
cudaMemset(devPtrB, 0, N * sizeof(float));

// call kernel
myKernel<<<...>>>(devPtrA, devPtrB, N);

cudaMemcpy(B, devPtrB, N * sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(devPtrA);
cudaFree(devPtrB);
...
}
Error Handling

All CUDA runtime API functions return an error code. The runtime maintains an error variable for each host thread that is overwritten by the error code every time an error concurs.

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaGetLastError()</td>
<td>Returns error variable and resets it to cudaSuccess</td>
</tr>
<tr>
<td>cudaGetErrorString()</td>
<td>Returns the message string from an error code</td>
</tr>
</tbody>
</table>

```c
cudaError_t err = cudaGetLastError();
if (cudaSuccess != err) {
    fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```
Porting Mandelbrot set fractal renderer to CUDA

- Source is in ~/tutorial/src2
  - fractal.c – reference C implementation
  - Makefile – make file
  - fractal.cu.reference – CUDA implementation for reference
Getting started

• ssh USER@ac.ncsa.uiuc.edu
• qsub -l -l walltime=02:00:00
• cd tutorial/src2
• make cpu
• ./fractal_cpu
• make convert

• copy fractal.bmp to your desktop
• display fractal.bmp on your desktop
Reference C Implementation

```c
void makefractal_cpu(unsigned char *image, int width, int height, double xupper,
    double xlower, double yupper, double ylower)
{
    int x, y;

    double xinc = (xupper - xlower) / width;
    double yinc = (yupper - ylower) / height;

    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            image[y*width+x] = iter((xlower + x*xinc), (ylower + y*yinc));
        }
    }
}
```
inline unsigned char iter(double a, double b) {
    unsigned char i = 0;
    double c_x = 0, c_y = 0;
    double c_x_tmp, c_y_tmp;
    double D = 4.0;

    while (((c_x*c_x+c_y*c_y) < D) && (i++ < 255)) {
        c_x_tmp = c_x * c_x - c_y * c_y;
        c_y_tmp = 2 * c_y * c_x;
        c_x = a + c_x_tmp;
        c_y = b + c_y_tmp;
    }

    return i;
}
CUDA Kernel Implementation

```c
__global__ void makefractal_gpu(unsigned char *image, int width, int height, double xupper, double xlower, double yupper, double ylower)
{
    int x = blockIdx.x;
    int y = blockIdx.y;

    int width = blockDim.x;
    int height = blockDim.y;

    double xupper=-0.74624, xlower=-0.74758, yupper=0.10779, ylower=0.10671;

    double xinc = (xupper - xlower) / width;
    double yinc = (yupper - ylower) / height;

    image[y*width+x] = iter((xlower + x*xinc), (ylower + y*yinc));
}
```
inline __device__ unsigned char iter(double a, double b)
{
    unsigned char i = 0;
    double c_x = 0, c_y = 0;
    double c_x_tmp, c_y_tmp;
    double D = 4.0;

    while (((c_x*c_x+c_y*c_y) < D) && (i++ < 255))
    {
        c_x_tmp = c_x * c_x - c_y * c_y;
        c_y_tmp = 2 * c_y * c_x;
        c_x = a + c_x_tmp;
        c_y = b + c_y_tmp;
    }

    return i;
}
Host Code

int width = 1024;
int height = 768;
unsigned char *image = NULL;
unsigned char *devImage;

image = (unsigned char*)malloc(width*height*sizeof(unsigned char));
cudaMalloc((void**)&devImage, width*height*sizeof(unsigned char));

dim3 dimGrid(width, height);
dim3 dimBlock(1);

makefractal_gpu<<<dimGrid, dimBlock>>>(devImage);

cudaMemcpy(image, devImage, width*height*sizeof(unsigned char), cudaMemcpyDeviceToHost);

free(image);
cudaFree(devImage);
Few Examples

- $x_{upper}=-0.74624$
- $x_{lower}=-0.74758$
- $y_{upper}=0.10779$
- $y_{lower}=0.10671$
- CPU time: 2.27 sec
- GPU time: 0.29 sec

- $x_{upper}=-0.754534912109$
- $x_{lower}=-.757077407837$
- $y_{upper}=0.060144042969$
- $y_{lower}=0.057710774740$
- CPU time: 1.5 sec
- GPU time: 0.25 sec
Sum reduction kernel example

• Source is in ~/tutorial/src4
• sum.c – reference C implementation
• Makefile – make file
• sum.cu.reference – CUDA implementation for reference
Sum reduction

```c
int main(int argc, char **argv)
{
    int i, N = 2097152;  // vector size
    double *A, s = 0.0f;

    A = (double*)malloc(N * sizeof(double));

    // generate random data
    for (i = 0; i < N; i++)
        A[i] = (double)rand()/RAND_MAX;

    s = sum(A, N);  // call compute kernel

    printf("sum=%.2fn", s);

    free(A);  // free allocated memory
}
```

```c
double sum(double* v, int n)
{
    int i;
    double s = 0.0f;

    for (i = 0; i < n; i++)
        s += v[i];

    return s;
}
```

\[ S = \sum_{k=0}^{n} \nu_k \]
Where do we find parallelism?

\[ S = \sum_{k=0}^{15} v_k \]
Where do we find parallelism?

N/2 additions can be done independently
Where do we find parallelism?

N/4 additions can be done independently.
Where do we find parallelism?

N/8 additions can be done independently
Where do we find parallelism?

N/16 additions can be done independently

\[ \mathbf{v_0} + \mathbf{v_1} \rightarrow \mathbf{S} \]
GPU kernel for N<=1024

```c
__global__ void sum (double *v)
{
    unsigned int t = threadIdx.x;
    unsigned int stride;

    for (stride = blockDim.x >> 1; stride > 0; stride >>= 1)
    {
        __syncthreads();
        if (t < stride)
            v[t] += v[t+stride];
    }
}

sum<<<1, N/2>>>(a);
```
The rest of the code

double *devPtrA; // allocate memory, copy data
cudaMalloc((void**)&devPtrA, N * sizeof(double));
cudaMemcpy(devPtrA, A, N * sizeof(double), cudaMemcpyHostToDevice);

sum<<<1, N/2>>>(devPtrA); // call compute kernel

cudaError_t err = cudaGetLastError(); // check for errors
if (cudaSuccess != err)
{
    fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err ));
    exit(EXIT_FAILURE);
}

// get results, free memory
cudaMemcpy(&s, devPtrA, sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(devPtrA);
Problems with this implementation

• $N \leq 1024$
  – A thread block may not have more than 512 threads

• Inefficient
  – Data is stored in global memory which has very high access latency

• $N$ must be a power of 2
Expanding to multiple thread blocks

N/2 additions can be done independently
Eliminating global memory access latency

N/2 additions can be done independently

Store partial sums in the per-block shared memory
Expanding to multiple thread blocks

N/4 additions can be done independently
Expanding to multiple thread blocks

N/8 additions can be done independently
Final sum reduction kernel

```c
__global__ void sum(double *v)
{
    extern double __shared__ sd[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;

    sd[tid] = v[i] + v[i+blockDim.x];
    __syncthreads();

    for (unsigned int s = blockDim.x/2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sd[tid] += sd[tid + s];
            __syncthreads();
        }
    }

    if (tid == 0) v[blockIdx.x] = sd[0];
}
```

- **perform first level of reduction, reading from global memory, writing to shared memory**
- **do reduction in shared memory**
- **write result for this block to global mem**
Are we done yet?

- We started with this

\[
\begin{array}{cccccccccccccccc}
V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 & V_9 & V_{10} & V_{11} & V_{12} & V_{13} & V_{14} & V_{15} \\
\end{array}
\]

- And ended with this

\[
\begin{array}{cc}
V_0 & V_1 \\
\end{array}
\]

- where \( v_0 \) and \( v_1 \) are partial sums computed by individual thread blocks, stored in global memory, and they still need to be added
- The final addition can be done by running the same kernel on this reduced data set
Modified host code

int threads = 64;
int old_blocks, blocks = N / threads / 2;
bloc\(s = (bloc\(s == 0) ? 1 : bloc\(s;\)
old_blocks = blocks;

while (blocks > 0)  // call compute kernel
{
    sum\(<<blocks, threads, threads*\text{sizeof(double)>>>>(devPtrA);
    old_blocks = blocks;
    blocks = blocks / threads / 2;
}

if (blocks == 0 && old_blocks != 1)  // final kernel call, if still needed
    sum\(<<1, old_blocks/2, old_blocks/2*\text{sizeof(double)>>>>(devPtrA);
Example run

• [kindr@ac src4]$ ./sum_cpu
  • Running CPU sum for 2097152 elements
  • sum=1048443.09
  • sec = 0.006771  GFLOPS = 0.309

• [kindr@ac src4]$ ./sum_gpu
  • Running GPU sum for 2097152 elements
  • Grid/thread dims are (16384), (64)
  • Grid/thread dims are (128), (64)
  • Grid/thread dims are (1), (64)
  • sum=1048443.09
  • sec = 0.000389  GFLOPS = 5.391