CSCI-GA.3033-008

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 8: Advanced Techniques

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Importance of Floating Points

- Many graphics operations are floating point operations
- GPU performance is measured in GFLOPS
Turing Award 1989 to William Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)
What is Excel doing?

A1: 1.33333333333333330000 =4/3
Floating Point

• We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 \times 10^9$

• Representation:
  - sign, exponent, mantissa: $(-1)^{\text{sign}} \times \text{mantissa} \times 2^{\text{exponent}}$
  - more bits for mantissa gives more accuracy
  - more bits for exponent increases range

• IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit mantissa
  - double precision: 11 bit exponent, 52 bit mantissa
IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit (called hidden 1 technique, except when exp = -127)
- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent
  - all 1s is largest exponent
  - bias of 127 for single precision and 1023 for double precision
  - summary: \((-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}\)

- Example:
  - decimal: -.75 = - ( \(\frac{1}{2} + \frac{1}{4}\) )
  - binary: -.11 = -1.1 \times 2^{-1}
  - floating point: exponent = 126 = 01111110

  - IEEE single precision: 10111111010000000000000000000000
More about IEEE floating Point Standard

Single Precision:

\[ (-1)^{\text{sign}} \times (1+\text{mantissa}) \times 2^{\text{exponent} - 127} \]

The variables shown in red are the numbers stored in the machine
Floating Point Example

what is the decimal equivalent of

1 01110110 10110000...0
Special Patterns

• Representation of zero
  – No hidden one
  – Exponent and mantissa are 0s

• When all exponent bits are ones
  – If mantissa is zero -> infinity
  – If mantissa is nonzero -> Not a Number (NaN)
What is the decimal equivalent of:

- $10011111110100000000000000000000$

So:
- Real exponent = 127 - 127 = 0
- There is hidden 1

$1.1010....0$

$= 1.625$

Final answer = -1.625
## In Summary About Floating Points

<table>
<thead>
<tr>
<th>exponent</th>
<th>mantissa</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>11...1</td>
<td>≠ 0</td>
<td>NaN</td>
</tr>
<tr>
<td>11...1</td>
<td>=0</td>
<td>((-1)^S \times \infty)</td>
</tr>
<tr>
<td>00...0</td>
<td>≠0</td>
<td>denormalized</td>
</tr>
<tr>
<td>00...0</td>
<td>=0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm Considerations

- Non representable numbers are rounded.
- This rounding error leads to different results depending on the order of operations.
  - Non-repeatability makes debugging harder.
- A common technique to maximize floating point arithmetic accuracy is to presort data before a reduction computation.
So..

When doing floating-point operations in parallel you have to decide:

• How much accuracy is good enough?
• Do you need single-precision or double precision?
• Can you tolerate presorting overhead, if you care about rounding errors?
Memory Alignment

• Memory access on the GPU works much better if the data items are aligned at 64 byte boundaries.
• Hence, allocating 2D arrays so that every row starts at a 64-byte boundary address will improve performance.
• Difficult to do for a programmer!
Pitch

Rows

Columns

Pitch

Padding
2D Arrays

- CUDA offers special versions of:
  - Memory allocation of 2D arrays so that every row is padded (if necessary). The function determines the best pitch and returns it to the program. The function name is `cudaMallocPitch()`.
  - Memory copy operations that take into account the pitch that was chosen by the memory allocation operation. The function name is `cudaMemcpy2D()`.
cudaMallocPitch( void** devPtr,
    size_t* pitch,
    size_t widthInBytes,
    size_t height)

• This allocates \textit{width} \times \textit{height} array.
• The value returned in pitch is the width in bytes of the allocation.
• The above function determines the best pitch and returns it to the program.
• It is strongly recommends the usage of this function for allocating 2D (and 3D) arrays.
cudaError_t cudaMemcpy2D ( void * dst,
    size_t dpitch,
    const void * src,
    size_t spitch,
    size_t width,
    size_t height,
    enum cudaMemcpyKind kind )

• dst - Destination memory address
• dpitch - Pitch of destination memory
• src - Source memory address
• spitch - Pitch of source memory
• width - Width of matrix transfer (in bytes)
• height - Height of matrix transfer (rows)
• kind - Type of transfer

the widths in memory in bytes including any padding added to the end of each row
Example

```c
int main(int argc, char * argv[]) {
    float * A, *dA;
    size_t pitch;

    A = (float *)malloc(sizeof(float)*N*N);
    cudaMallocPitch(&dA, &pitch, sizeof(float)*N, N);

    //copy memory from unpadded array A of 760 by 760 dimensions
    //to more efficient dimensions on the device
    cudaMemcpy2D(dA, pitch, A, sizeof(float)*N, sizeof(float)*N, N, 
                  cudaMemcpyHostToDevice);
    ...
}
```
So..

Pitch is a good technique to speedup memory access
• There are two drawbacks that you have to live with:
  • Some wasted space
  • A bit more complicated elements access
Streams

• A sequence of operations that execute on the device in the order in which they are issued by the host code
• Operations in different streams can be interleaved and, when possible, they can even run concurrently.
• A stream can be sequence of kernel launches and host-device memory copies
• Can have several open streams to the same device at once
• Need GPUs with concurrent transfer/execution capability
• Potential performance improvement: can overlap transfer and computation
Streams

• By default all transfers and kernel launches are assigned to stream 0
  – This means they are executed in order
Example: Default Stream

```c
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- In the code above, from the perspective of the device, all three operations are issued to the same (default) stream and will execute in the order that they were issued.
- From the perspective of the host:
  - data transfers are blocking or synchronous transfers
  - kernel launch is asynchronous.

```c
anyCPUfunction();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```
Example: Non-Default Stream

Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code as follows:

```c
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

To issue data transfer to non-default stream (non-blocking):

```c
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

To launch a kernel to non-default stream:

```c
increment<<<1,N,0,stream1>>>(d_a);
```
Important

- All operations in non-default streams are non-blocking with respect to the host code.
- Sometimes you need to synchronize the host code with operations in a stream.
- You have several options:
  - `cudaDeviceSynchronize()` → blocks host
  - `cudaStreamSynchronize(stream)` → blocks host
  - `cudaStreamQuery(stream)` → does not block host
Streams

The amount of overlap execution between two streams depends on:

- Device supports overlap transfer and kernel execution
- Devices supports concurrent kernel execution
- Device supports concurrent data transfer
- The order on which commands are issued to each stream
Using streams to overlap device execution with data transfer

• Conditions to be satisfied first:
  – The device must be capable of concurrent copy and execution.
  – The kernel execution and the data transfer to be overlapped must both occur in different, non-default streams.
  – The host memory involved in the data transfer must be pinned memory.
Using streams to overlap device execution with data transfer

for (int i = 0; i < nStreams; ++i) {

    int offset = i * streamSize;

    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);

    kernel<<<>(d_a, offset);

    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}

So..

- Streams are a good way to overlap execution and transfer, hardware permits.
- Don’t confuse kernels, threads, and streams.
Pinned Pages

- Allocate page(s) from system RAM (cudaMallocHost() or cudaHostAlloc())
  - Accessible by device
  - Cannot be page out
  - Enables highest memory copy performance (cudaMemcpyAsyc())

- If too much pinned pages, overall system performance may greatly suffer.
So..

- If the CPU program requires a lot of memory, then pinned pages is not a good idea.
Texture Memory

To accelerate frequently performed operations such as mapping a 2D "skin" onto a 3D polygonal model.
Texture Memory
Texture Memory

Capabilities:

- Ability to cache global memory
- Dedicated interpolation hardware
- Provides a way to interact with the display capabilities of the GPU.

The best performance is achieved when the threads of a warp read locations that are close together from a spatial locality perspective.
Texture Memory

- Read only and cached
- The texture cache is optimized for 2D spatial locality.
- Part of DRAM
- The process of reading a texture is called a *texture fetch*.
- Can be addressed as 1D, 2D, or 3D dimensional arrays.
- Elements of the array are called *texels*. 
So..

• R/O no structure $\rightarrow$ constant memory

• R/O array structured $\rightarrow$ texture memory
Asynchronous Execution

• Asynchronous = returns to host right-away and does not wait for device
• This includes:
  – Kernel launches;
  – Memory copies between two addresses to the same device memory;
  – Memory copies from host to device of a memory block of 64 KB or less;
  – Memory copies performed by functions that are suffixed with Async;
  – Memory set function calls.
Asynchronous Execution

- Some CUDA API calls and all kernel launches are asynchronous with respect to the host code.
- This means error-reporting is also asynchronous.
- Asynchronous transfer (cudaMemcpyAsync() ) version requires pinned host memory
- On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations.
Asynchronous Execution

cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernelpa<<grid, block>>>(a_d);
cpuFunction();
Other Sources of Concurrency

- Some devices of compute capability 2.x and higher can execute multiple kernels concurrently.
- The maximum number of kernel launches that a device can execute concurrently is 32 on devices of compute capability 3.5 and 16 on devices of lower compute capability.
- A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.
- Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.
- Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page locked host memory.
Some Useful Tools
Some nvcc features: --ptxas-options=--v
  - Print the smem, register and other resource usages

Generates CUDA binary file: nvcc -cubin
  - cubin file is the cuda executable
  - The default for nvcc is to embed it in the host executable
Dealing with binary files

<table>
<thead>
<tr>
<th>Extract ptx and extract and disassemble cubin from the following input files:</th>
<th>cuobjdump</th>
<th>nvdisasm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host binaries</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Executables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External fatbinary files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control flow analysis and output</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Advanced display options</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
nvprof

• CUDA profiler

$ nvprof [nvprof_args] <app> [app_args]

• To profile a region of the application:
  1. #include <cuda_profiler_api.h>
  2. in the host function surround the region with:
     • cudaProfilerStart()
     • cudaProfilerStop()
  3. nvcc myprog.cu
  4. nvprof --profile-from-start-off ./a.out
nvprof summary mode (default)

```bash
$ nvprof dct8x8

=*=*=*=* Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>49.52</td>
<td>9.36ms</td>
<td>101</td>
<td>92.68us</td>
<td>92.31us</td>
<td>94.31us</td>
<td>CUDAkernel2DCT(float*, float*, int)</td>
</tr>
<tr>
<td>37.47</td>
<td>7.08ms</td>
<td>10</td>
<td>708.31us</td>
<td>707.99us</td>
<td>708.50us</td>
<td>CUDAkernel1DCT(float*,int, int,int)</td>
</tr>
<tr>
<td>3.75</td>
<td>708.42us</td>
<td>1</td>
<td>708.42us</td>
<td>708.42us</td>
<td>708.42us</td>
<td>CUDAkernel1IDCT(float*,int,int,int)</td>
</tr>
<tr>
<td>1.84</td>
<td>347.99us</td>
<td>2</td>
<td>173.99us</td>
<td>173.59us</td>
<td>174.40us</td>
<td>CUDAkernelQuantizationFloat()</td>
</tr>
<tr>
<td>1.75</td>
<td>331.37us</td>
<td>2</td>
<td>165.69us</td>
<td>165.67us</td>
<td>165.70us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>1.41</td>
<td>266.70us</td>
<td>2</td>
<td>133.35us</td>
<td>89.70us</td>
<td>177.00us</td>
<td>[CUDA memcpyHtoD]</td>
</tr>
<tr>
<td>1.00</td>
<td>189.64us</td>
<td>1</td>
<td>189.64us</td>
<td>189.64us</td>
<td>189.64us</td>
<td>CUDAkernelShortDCT(short*, int)</td>
</tr>
<tr>
<td>0.94</td>
<td>176.87us</td>
<td>1</td>
<td>176.87us</td>
<td>176.87us</td>
<td>176.87us</td>
<td>[CUDA memcpyHtoA]</td>
</tr>
<tr>
<td>0.92</td>
<td>174.16us</td>
<td>1</td>
<td>174.16us</td>
<td>174.16us</td>
<td>174.16us</td>
<td>CUDAkernelShortIDCT(short*, int)</td>
</tr>
<tr>
<td>0.76</td>
<td>143.31us</td>
<td>1</td>
<td>143.31us</td>
<td>143.31us</td>
<td>143.31us</td>
<td>CUDAkernelQuantizationShort(short*)</td>
</tr>
<tr>
<td>0.52</td>
<td>97.75us</td>
<td>1</td>
<td>97.75us</td>
<td>97.75us</td>
<td>97.75us</td>
<td>CUDAkernel2IDCT(float*, float*)</td>
</tr>
<tr>
<td>0.12</td>
<td>22.59us</td>
<td>1</td>
<td>22.59us</td>
<td>22.59us</td>
<td>22.59us</td>
<td>[CUDA memcpyDtoA]</td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2012
nvprof trace mode

```
$ nvprof --print-gpu-trace dct8x8

<table>
<thead>
<tr>
<th>Start</th>
<th>Duration</th>
<th>Grid Size</th>
<th>Block Size</th>
<th>Regs</th>
<th>SSMem</th>
<th>DSMem</th>
<th>Size</th>
<th>Throughput</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>167.82ms</td>
<td>176.84us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.05MB</td>
<td>5.93GB/s</td>
<td>[CUDA memcpy HtoA]</td>
</tr>
<tr>
<td>168.00ms</td>
<td>708.51us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>168.95ms</td>
<td>708.51us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>169.74ms</td>
<td>708.26us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>170.53ms</td>
<td>707.89us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>171.32ms</td>
<td>708.12us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>172.11ms</td>
<td>708.05us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>172.89ms</td>
<td>708.38us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>173.68ms</td>
<td>708.31us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>174.47ms</td>
<td>708.15us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>175.26ms</td>
<td>707.95us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>176.05ms</td>
<td>173.87us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>27</td>
<td>0B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernelQuantization (...)</td>
</tr>
<tr>
<td>176.23ms</td>
<td>22.82us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.05MB</td>
<td>45.96GB/s</td>
<td>[CUDA memcpyDtoA]</td>
</tr>
</tbody>
</table>
```
Print individual kernel invocations and sort them in chronological order.

Print CUDA runtime/driver API trace.

```plaintext
nvprof --print-gpu-trace --print-api-trace dct8x8

Profiling result:

<table>
<thead>
<tr>
<th>Start</th>
<th>Duration</th>
<th>Grid Size</th>
<th>Block Size</th>
<th>Regs</th>
<th>SSMem</th>
<th>DSMem</th>
<th>Size</th>
<th>Throughput</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>167.82ms</td>
<td>176.84us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.05MB</td>
<td>5.93GB/s</td>
<td>[CUDA memcpy HtoA]</td>
</tr>
<tr>
<td>167.81ms</td>
<td>2.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>167.81ms</td>
<td>38.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>167.85ms</td>
<td>1.00ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaDeviceSynchronize</td>
</tr>
<tr>
<td>168.00ms</td>
<td>708.51us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
<tr>
<td>168.86ms</td>
<td>2.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaConfigureCall</td>
</tr>
<tr>
<td>168.86ms</td>
<td>1.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>168.86ms</td>
<td>1.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>168.86ms</td>
<td>1.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>168.87ms</td>
<td>0ns</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>168.87ms</td>
<td>24.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>168.89ms</td>
<td>761.00us</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>cudaDeviceSynchronize</td>
</tr>
<tr>
<td>168.95ms</td>
<td>708.51us</td>
<td>(64 64 1)</td>
<td>(8 8 1)</td>
<td>28</td>
<td>512B</td>
<td>0B</td>
<td>-</td>
<td>-</td>
<td>CUDAkernel1DCT(float*, ...)</td>
</tr>
</tbody>
</table>
```
nvcc --devices 0 --query-events ./a.out

• Gives very useful information, such as:
  – number of global memory loads, stores, …
  – number of global memory coalesced
  – branch divergences
  – …

• You must specify the event:

  $ nvprof --devices 0 --events branch,divergent_branch dct8x8
Conclusions

• There are many performance enhancement techniques in our arsenal:
  – Alignment
  – Streams
  – Pinned pages
  – Texture memory
  – Asynchronous execution

• If your program is making use of a lot of FP operations, be careful about rounding errors.

• There are tools to help you!