GPU Architecture and CUDA Programming

Chris Kauffman

Last Updated:
Mon Nov 22 09:40:15 AM CST 2021

Logistics

Reading

GPU parallel program development using CUDA by Tolga Soyata

- Ch 6 start GPU Coverage
- UMN Library Link

A2 Out

- Take a quick tour
- Arriving soon: testing files for Problems 2/3 + optional Problem 4
- Note on SSH + MPI stalls

Poll on Final Exam

Will poll on Final Exam options over the next 5 days

- Option A: Mini-exam 4 (10%) + Final Exam (10%)
- Option B: Final Exam Last Day of class (20%)

Today

GPUs will Feel Different

Distributed / Threaded Programming

- Most effective strategies looked for ways to assign lots of work to limited number of procs/threads
- Poo-pooed the idea of "Assume length N array and N processors", too impractical

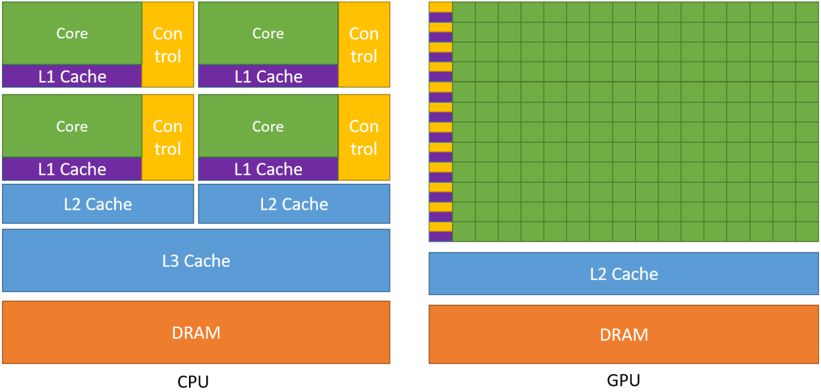
GPU Programming

- Threads are essentially cost-free, close to theoretical models so...
  Assume length N array and N processors. It's actually practical and beneficial.
- Will require some mental adjustment

GPUs are a Co-Processor or Accelerator

- CPU is still in charge, has access to main memory
- GPU is a partner chip, has a distinct set of memory
- Sections of code will feel like Distributed architecture
  - CPU / GPU memory transfers
  - Barriers / synchronization as CPU waits for GPU to finish
- GPU itself is like a multicore system on steroids

CPU vs GPU



Source: NVidia Docs "CUDA C++ Programming Guide"

- GPU cores are simpler, slower, but there are TONs of them
- GPU has its own memory hierarchy: cache and DRAM
- Requires explicit transfers to/from CPU

Why do GPUs Look like this?

140 ■ GPU Parallel Program Development Using CUDA

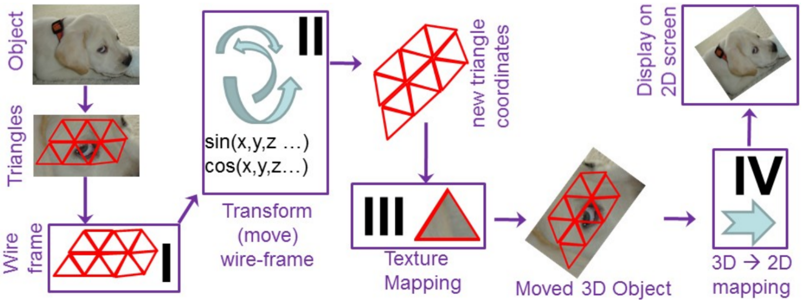


FIGURE 6.2 Steps to move triangulated 3D objects. Triangles contain two attributes: their *location* and their *texture*. Objects are moved by performing mathematical operations only on their coordinates. A final texture mapping places the texture back on the moved object coordinates, while a 3D-to-2D transformation allows the resulting image to be displayed on a regular 2D computer monitor.

Source: GPU parallel program development using CUDA by Tolga Soyata, 2018. (UMN Library Link)

CUDA : NVidia's General Purpose GPU Technology

- Games exploit GPU capabilities for parallelism via specialized graphics libraries like OpenGL
  - Oriented specifically towards graphics operations
  - Vendor like NVidia provides their OpenGL library which accelerates graphics processing
- Researchers wanted to exploit the massively parallel FP operations in GPUs to speed simulations (circa year 2000)
  - Started reverse engineering physics simulations to present them as Graphics problems
  - Achieved tremendous speedup but it was a pain to code
- NVidia recognized the new market for their chips, began exposing GPU capabilities for other applications: GPGPU is a General Purpose GPU
  - CUDA version 1 released 2007
  - Provides GPU capabilities through Threads
  - Provides a C/C++ code interface to run "kernel" functions on the GPU with many threads

CUDA Terminology

Thread A set of operations; can be as small as a single addition; each thread has identifying information (index, # of other threads)

Kernel A function which expresses what a thread should do. Many Threads execute the same Kernel code but can operate on different data based on their Thread index.

Block A group of executing threads which can share some local memory

Execution Context Parameters for a Kernel run indicating number of Blocks, Threads per Block, and amount of shared memory

Host The CPU, sets Execution Context, launches Kernels on GPU, waits for results.

Device The GPU which runs Kernels on tons of threads

Hello CUDA

```
// hello.cu: C code demonstrating basics of cuda

#include <stdio.h>

__global__ void hello_gpu() {   // __global__ => called from CPU/GPU,
  printf("Block %02d Thread %02d: Hello World\n",     //  runs on GPU
         blockIdx.x,            // ever-present structs which gives
         threadIdx.x);          // each GPU thread indexing info
}

int main (int argc, char *argv[]){
  printf("CPU: Running 1 block w/ 16 threads\n");
  hello_gpu<<<1,16>>>();        // executes in 1 block, 16 threads per block
  cudaDeviceSynchronize();      // ensures GPU completes operations

  printf("\n");

  int nblocks  = argc < 2 ? 3 : atoi(argv[1]); // default 3 blocks
  int nthreads = argc < 3 ? 4 : atoi(argv[2]); // default 4 threads/block
  printf("CPU: Running %d blocks w/ %d threads\n",
         nblocks, nthreads);

  hello_gpu<<<nblocks, nthreads>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Compiling and Running Code

```
# log into the veggie cluster for access to an NVidia GPU
val [~]% ssh csel-broccoli.cselabs.umn.edu

# check for presence of nvidia hardware
csel-broccoli [~]% lspci | grep -i nvidia
3b:00.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)

csel-broccoli [~]% cd 14-gpu-cuda-code

# load CUDA tools on CSE Labs
csel-broccoli [14-gpu-cuda-code]% module load soft/cuda

# nvcc is the CUDA compiler - C++ syntax, gcc-like behavior
csel-broccoli [14-gpu-cuda-code]% nvcc hello.cu

# run with defaults
csel-broccoli [14-gpu-cuda-code]% ./a.out
CPU: Running 1 block w/ 16 threads
Block 00 Thread 00: Hello World
Block 00 Thread 01: Hello World
...
Block 00 Thread 15: Hello World

CPU: Running 3 blocks w/ 4 threads
Block 00 Thread 00: Hello World
Block 00 Thread 01: Hello World
Block 00 Thread 02: Hello World
Block 00 Thread 03: Hello World
Block 02 Thread 00: Hello World
...
```

Low-level Contents of CUDA Executables

```
>> module load soft/cuda            # load tools
>> nvcc hello.cu                    # ncompile code

>> file a.out                       # show file type of executable
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
... for GNU/Linux 3.2.0, not stripped

>> readelf -S a.out | grep -i nv    # search for special ELF sections
  [17] .nv_fatbin        PROGBITS         000000000007f4f0  0007f4f0
  [18] __nv_module_id    PROGBITS         00000000000805c8  000805c8
  [29] .nvFatBinSegment  PROGBITS         000000000009e058  0009d058
```

- Compiled CUDA programs are ELF format executable
- Standard sections present like `.text` with host instructions (x86-64) and global data `.data`, `.bss` etc.
- Additional sections contain a *nested ELF file* with GPU code in PTX, the Assembly language used in NVidia GPUs

PTX: CUDA Assembly Language

- PTX: Parallel Thread Execution, VM instructions for the GPU
- Converted on the fly to GPU execution, can use inline PTX

```
>> cuobjdump a.out -sass -ptx       # disassemble CUDA portion of exec
...                                 # show GPU PTX assembly instructions
Fatbin elf code:
================
arch = sm_52
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit

code for sm_52
        Function : _Z9hello_gpuv
.headerflags    @"EF_CUDA_SM52 EF_CUDA_PTX_SM(EF_CUDA_SM52)"
                                                                  /* 0x001c4400fe0007f6 */
/*0008*/                   MOV R1, c[0x0][0x20] ;                 /* 0x4c98078000870001 */
/*0010*/         {         IADD32I R1, R1, -0x8 ;                 /* 0x1c0ffffff870101 */
/*0018*/                   S2R R3, SR_TID.X         }
                                                                  /* 0xf0c8000002170003 */
                                                                  /* 0x001fd000e22007f0 */
/*0028*/         {         MOV32I R4, 0x0 ;                       /* 0x010000000007f004 */
/*0030*/                   S2R R2, SR_CTAID.X         }
...
```

Link: cuobjdump Documentation

I'm Not Fat, I'm Just full of Code

CUDA Executable are "Fat" binaries - may contain multiple embedded ELF files to support several GPU versions

```
>> nvcc hello.cu                # compile with defaults

>> cuobjdump a.out -lelf        # list embedded ELF files
ELF file    1: a.1.sm_52.cubin
ELF file    2: a.2.sm_52.cubin

# compile with specific CUDA version support embedded
>> nvcc hello.cu -gencode arch=compute_52,code=sm_52 \
                 -gencode arch=compute_70,code=sm_70

# list embedded ELF files pertaining to CUDA
>> cuobjdump a.out -lelf
ELF file    1: a.1.sm_52.cubin
ELF file    2: a.2.sm_70.cubin
ELF file    3: a.3.sm_52.cubin
ELF file    4: a.4.sm_70.cubin
```

Fat executables are not novel, have been used by Apple in transition periods every time they change their mind about processor architecture

CUDA is Advancing 1 / 2

CUDA is a **rapidly** advancing in technology with frequent changes.


CUDA now supports `printf`s directly in the kernel. For formal description see Appendix B.16 of the CUDA C Programming Guide.

77

Share Edit Follow

edited Oct 25 '17 at 13:37
shookees
358 ●2 ●6 ●18

answered Jul 5 '11 at 17:10
M. Tibbits
8,113 ●7 ●41 ●58

12 I think the link is not pointing to the right place anymore. Here is an alternate link: docs.nvidia.com/cuda/cuda-c-programming-guide/... – cyang Jan 28 '13 at 0:55

13 Note: "now" means compute capability 2.x or higher. – colgur Feb 22 '13 at 16:08

Source: SO 'printf inside CUDA global function'

Note the mention of **Compute Capability** which refers to the version of CUDA supported by GPU hardware; version reported via

- Utilities like `nvidia-smi` or
- Programmatically within CUDA (see device query example)

CUDA is Advancing 2 / 2

5.4.1. Arithmetic Instructions

Table 3 gives the throughputs of the arithmetic instructions that are natively supported in hardware for devices of various compute capabilities.

Table 3. Throughput of Native Arithmetic Instructions. (Number of Results per Clock Cycle per Multiprocessor)

<table>
<tr><th rowspan="2"></th><th colspan="9">Compute Capability</th></tr>
<tr><th>3.5, 3.7</th><th>5.0, 5.2</th><th>5.3</th><th>6.0</th><th>6.1</th><th>6.2</th><th>7.x</th><th>8.0</th><th>8.6</th></tr>
<tr><td>16-bit floating-point add, multiply, multiply-add</td><td colspan="2">N/A</td><td>256</td><td>128</td><td>2</td><td>256</td><td>128</td><td colspan="2">256[3]</td></tr>
<tr><td>32-bit floating-point add, multiply, multiply-add</td><td>192</td><td colspan="2">128</td><td>64</td><td colspan="2">128</td><td colspan="2">64</td><td>128</td></tr>
<tr><td>64-bit floating-point add, multiply, multiply-add</td><td>64[4]</td><td colspan="2">4</td><td>32</td><td colspan="2">4</td><td>32[5]</td><td>32</td><td>2</td></tr>
<tr><td>32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)</td><td colspan="3">32</td><td>16</td><td colspan="2">32</td><td colspan="3">16</td></tr>
<tr><td>32-bit integer add, extended-precision add, subtract, extended-precision subtract</td><td>160</td><td colspan="2">128</td><td>64</td><td colspan="2">128</td><td colspan="3">64</td></tr>
<tr><td>32-bit integer multiply, multiply-add, extended-precision multiply-add</td><td>32</td><td colspan="5">Multiple instruct.</td><td colspan="3">64[6]</td></tr>
</table>

Source: NVidia CUDA Toolkit Documentation, v11.5

Doing Work in CUDA

1. Transfer data from CPU (host) to GPU (device)
2. Launch Kernels to compute results on GPU in parallel
3. Transfer results from GPU (device) back to CPU (host)

#2 above can be "looped"

`vecadd_cuda.cu` Demo

- Demonstrates transfer to/from GPU
- Simple kernel to do element-wise addition in an array

Device Memory Allocation / De-Allocation

```
// vecadd_cuda.cu
int main(){
  ...;
  // allocate device (GPU) memory
  float *dev_x, *dev_y, *dev_z;
  cudaMalloc((void**) &dev_x, length * sizeof(float));
  cudaMalloc((void**) &dev_y, length * sizeof(float));
  cudaMalloc((void**) &dev_z, length * sizeof(float));
  ...;
  // free device memory
  cudaFree(dev_x); cudaFree(dev_y); cudaFree(dev_z);
  ...
}
```

- Similar semantics to `malloc()` / `free()`
- `cudaMalloc()` returns `int` with success as `CUDA_SUCCESS`

Data Transfer Between Host / Device

```
// vecadd_cuda.cu
int main(){
  ...;
  // copy host memory to device
  cudaMemcpy(dev_x, host_x, length*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_y, host_y, length*sizeof(float), cudaMemcpyHostToDevice);
  ...;

  // do some work here

  // copy device memory to host
  cudaMemcpy(host_z, dev_z, length*sizeof(float), cudaMemcpyDeviceToHost);
  ...;
}
```

- Like distributed memory send / receive
- Copying memory GPU $\rightarrow$ CPU always blocks CPU
  - GPU / CPU work independently (asynchronously)
  - Memory transfer induces a sync point: CPU waits for launched kernels to complete, transfer of data
- It is possible to create memory maps between host/device to automate this, may discuss later

Kernel Launch

```
// vecadd_cuda.cu
int main(){
  ...;
  // calculate params for kernel execution
  long nthreads = 256;                         // fixed number of threads/block
  long nblocks  = (length+255) / nthreads; // ensure sufficient blocks to
                                             // cover whole array
  printf("Running %ld Blocks w/ %ld threads each\n",
         nblocks, nthreads);

  // execute the GPU kernel
  vector_add<<<nblocks, nthreads>>>(length, dev_x, dev_y, dev_z);
  ...;
}
```

- Algorithm assumes 1 thread per array element
- Threads always launched in blocks w/ identical # of threads
- Must ensure enough blocks $\times$ threads created to cover array
- May lead to "extra" threads : handle this in kernel

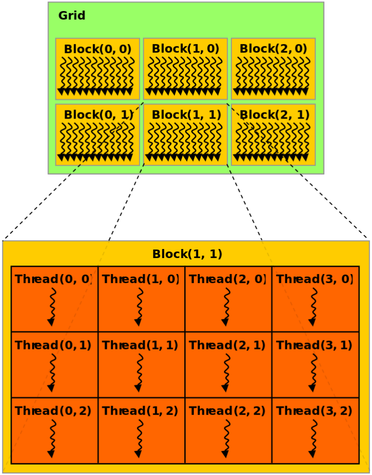
Kernel Code

```
// vecadd_cuda.cu
// KERNEL: each thread performs one pair-wise addition
__global__ void vector_add(long length,
                           float* x, float* y, float* z)
{
  long idx = threadIdx.x + blockDim.x * blockIdx.x;
  if(idx < length){
    z[idx] = x[idx] + y[idx];
  }
}
```

- Each thread handles 1 addition
- Index calculated using variables `threadIdx, blockDim`; several pre-defined variables like this in CUDA

  ```
  threadIdx.x  // x-index of thread within block
  blockDim.x   // x-dim (width) of thread's block
  blockIdx.x   // x-index of thread's block within grid
  gridDim.x    // x-dim (width) of the thread's grid
  // x/y/z fields available for all of these
  ```
- Note conditional which excludes "excess" threads

Threads in Blocks in Grids



Source: Wikip "Threaded Block (CUDA)"

CUDA grouping is

- Thread (`threadIdx`) in Block (`blockDim`)
- Block (`blockIdx`) in Grid (`gridDim`)

Memory

- Threads in the same Block can Share local/fast Memory (cache)
- All threads can access Global GPU Memory

Likely we will only deal with Threads + Blocks as they are enough trouble

Exercise: Array Summing

- Consider summing an array stored on the CPU
- Describe basic steps to do execute this on the GPU
- How is this problem different from the `vector_add()` version
- What makes it trickier?