GPU Architecture and CUDA Programming

Chris Kauffman

Last Updated:
Wed Nov 17 08:17:00 AM CST 2021

Logistics

Reading

GPU parallel program development using CUDA by Tolga Soyata

- Ch 6 start GPU Coverage
- UMN Library Link

Assignments

Will release A2 Friday

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

- Begin GPU / CUDA coverage
- Focus on orientation

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"

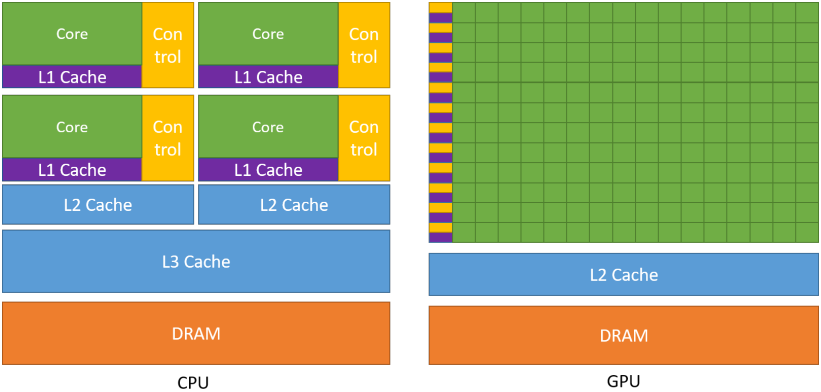
GPU Programming

- Threads are essentially cost-free, close to theoretical models so...
- Assume length N array and N processors
- 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, requires explicit transfers from/to CPU

Why do GPUs Look like this?

140 ■ GPU Parallel Program Development Using CUDA

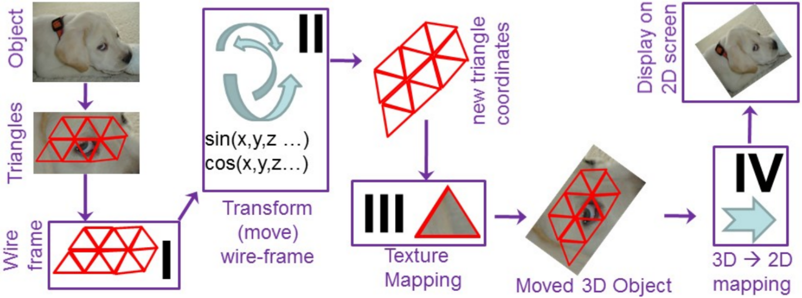


FIGURE 6.2 Steps to move triangulated 3D objects. Triangles contain two attributes: their *location* and their *texture*. Objects are moved by performing mathematical operations only on their coordinates. A final texture mapping places the texture back on the moved object coordinates, while a 3D-to-2D 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 General Purpose applications (GPGPU)
  - 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

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

#include <stdio.h>

__global__ void hello_gpu() {   // __global__ => called from CPU/GPU, runs on G
  printf("Block %02d Thread %02d: Hello World\n",
         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 - must have 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 Programs

```
>> 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)
- Embedded in them are sections of 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

If time permits...

- Examine device query example from NVidia
- Start looking at `vecadd_cuda.cu` for transfer operations