GPU Architecture and CUDA Programming

Chris Kauffman

Last Updated:
Mon Nov 29 09:40:53 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

Agenda

- Mon: Wrap GPU Discussion
- Wed: Guest Lecture on Sparse Distributed operations for fluid dynamics

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

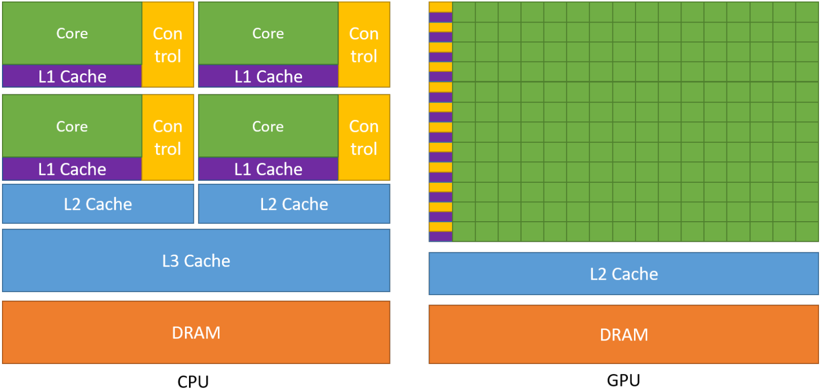
GPU Programming

- 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

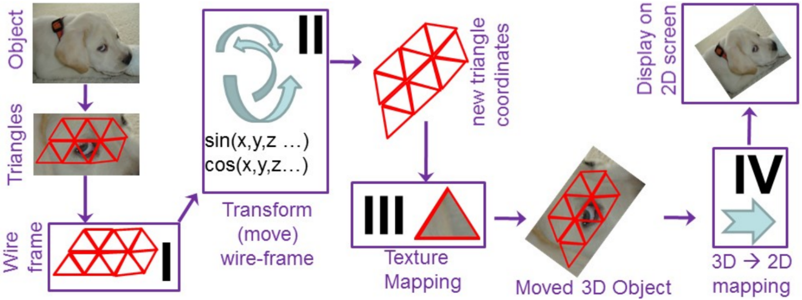


FIGURE 6.2 Steps to move triangulated 3D objects. Triangles contain two attributes: their *location* and their *texture*. Objects are moved by performing mathematical operations only on their coordinates. A final texture mapping places the texture back on the moved object coordinates, while a 3D-to-2D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

77

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

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 (<code>__log2f</code>), base 2 exponential (<code>exp2f</code>), sine (<code>__sinf</code>), cosine (<code>__cosf</code>)</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 × threads created to cover array
- May lead to “extra” threads : handle this in kernel

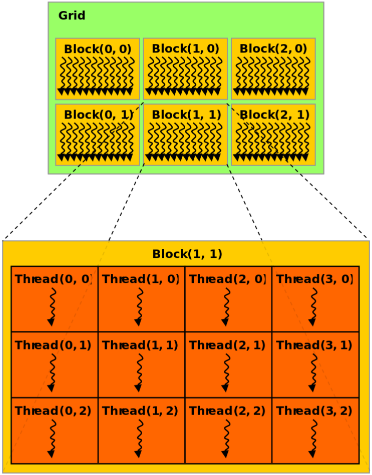
Kernel Code

```
// 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

Repeated Kernel Invocation has Overhead 1 / 2

GPU threads perfectly capable of iteration, often better to launch a single Kernel that loops than repeatedly launching a kernel

```
// vecloop_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];
  }
}
// KERNEL: each thread performs a loop of additions
__global__ void vector_loopadd(long iters, long length, float* x, float* y, float* z) {
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if(idx < length){
    for(long i=0; i<iters; i++){
      z[idx] = x[idx] + y[idx];
    }
  }
}
int main(int argc, char *argv[]){
  ...;
  for(long i=0; i<iterations; i++){
    vector_add<<<nblocks, nthreads>>>(length, dev_x, dev_y, dev_z);
  }
  ...;
  vector_loopadd<<<nblocks, nthreads>>>(iterations, length, dev_x, dev_y, dev_z);
```

Repeated Kernel Invocation has Overhead 2 / 2

```
broccoli>> nvcc vecloop_cuda.cu

# repeatedly launch kernel from host
broccoli>> time ./a.out 1000000 9000 host > /dev/null
real    0m1.079s
user    0m0.750s
sys     0m0.305s

# loop on device within kernel
broccoli>> time ./a.out 1000000 9000 device > /dev/null
real    0m0.686s
user    0m0.451s
sys     0m0.214s
```

Lesson: if computation allows for iteration, do so on GPU

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?

Answers: Array Summing

- Same basic steps
  - Transfer data to GPU
  - Execute summing kernel
  - Transfer answer back to CPU
- Each thread has little work
- Primary work is a **Reduction** which requires synchronization between thread and blocks

Array Sum: Naive vs Synchronization

```
// arraysum_cuda.cu

// all threads hit the same global sum; no syncronization on global
// memory so results are not computed correctly
__global__ void array_sum_1(int length, float* data, float *sum)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i < length){
    float myelem = data[i];
    *sum += mylelem;            // unsynced add to sum
  }
}

// all threads hit the same global sum with atomic operations
__global__ void array_sum_2(int length, float* data, float *sum)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i < length){
    float myelem = data[i];
    atomicAdd(sum, myelem);     // safe add to sum
  }
}
```

- `array_sum_1()` is incorrect due to race conditions
- `array_sum_2()` is correct but slow

CUDA Atomic Operations

- All threads can access GPU global memory but it is NOT synchronized
- CUDA Atomic Operations[1] like `atomicAdd()` are guaranteed to avoid race conditions between threads
- Variety of ops provided including arithmetic, bitwise ops, and compare + exchange operations

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Speeding up Reductions

- NVIDIA has its own presentation[2] on fast reductions
- It's a tricky business as GPU is better at embarrassingly parallel execution and CUDA is tailored to this
- We will touch on a few aspects but to demonstrate different aspects CUDA techniques but won't strive for perfection

[2] https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Block Shared Memory

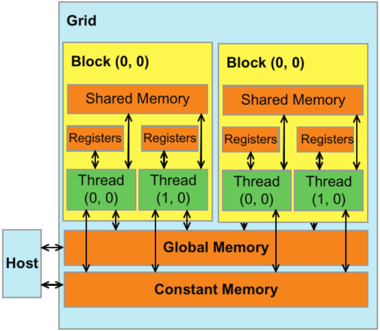


FIGURE 5.2

Overview of the CUDA device memory model.

Source: Programming Massively Parallel Processors by Kirk and Hwu

CUDA allows explicit control over cache memory shared among threads in block via `__shared__` keyword

```
__global__ void some_kernel(...){
{
  __shared__ float blockvals[256];
  // stored in cache, all threads in
  // block can access the array
  ...;
```

By default must use compile-time constant sizes for shared arrays

Synchronizing Threads

- Blocks of Threads will not all run in parallel
- Usually a **Warp** of 32 threads is run together
- Means some threads in a block may execute before others
- Presents a problem for shared memory
- `__syncthreads();` used as a Barrier for threads, guarantees all complete one set of operations

```
// nonsense example of shared memory + synchronization
__global__ void some_kernel(...){
{
  __shared__ int blockvals[256]; // shared data in cache
  int tid = threadIdx.x;

  blockvals[tid] = tid;         // all threads assign to blockvals

  __syncthreads();              // barrier to ensure all threads assign
                                // to blockvals before proceeding to...

  if(tid < 256-2){
    int mysum =
      blockvals[tid+0]+         // depends on blockvals[] being filled
      blockvals[tid+1]+         // by all threads
      blockvals[tid+2];
    ...;
```

Dynamically Allocating Shared Memory

When using shared memory, often want size dependent on number of threads

Statically Allocated

```
// static allocation of shared block
#define NTHREADS 64

__global__ void some_kern(...){
{
  __shared__ int blockvals[NTHREADS];
  ...
}
int main(...){


  some_kern<<<nblocks, NTHREADS>>>(..);
  ...;
}
```

Can use static size for shared memory + pre-defined number of threads

Dynamically Allocated

```
// dynamic allocation of shared block


__global__ void some_kern(...){
{
  extern __shared__ int blockvals[];
  ...
}
int main(...){
  int nthreads = ...;
  size_t shared_size = nthreads*sizeof(float);
  some_kern<<<nblocks, nthreads, shared_size>>>(..);
  ...;
}
```

Kernel Invocation can include size of shared memory, kernel declares with `extern` keyword

Exercise: Compare Kernels

```
__global__ void array_sum_3(int length, float* data, float *sum) {
  if(threadIdx.x == 0){
    float blocksum = 0.0;
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    for(int i=0; i < blockDim.x; i++){
      if(idx+i >= length){
        break;
      }
      blocksum += data[i+idx];
    }
    atomicAdd(sum, blocksum);
  }
}

__global__ void array_sum_4(int length, float* data, float *sum) {
  extern __shared__ float blockvals[];
  blockvals[threadIdx.x] = 0.0;
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if(idx < length){
    blockvals[threadIdx.x] = data[idx];
  }
  __syncthreads();
  if(threadIdx.x == 0){
    float blocksum = 0.0;
    for(int i=0; i < blockDim.x; i++){
      blocksum += blockvals[i];
    }
    atomicAdd(sum, blocksum);
  }
}
```

Describe the differences between these two kernels. Predict which is speedier.

Answers: Compare Kernels

- `array_sum_3()` simply has Thread 0 sum some array elements in a local variable (register) and then `atomicAdd()` to the global sum
- `array_sum_4()` has all threads load elements into a shared array
- Leads to cached data
- MUST synchronize threads prior to moving ahead to ensure all elements loaded into the array
- Thread 0 then iterates through this array summing and doing a final `atomicAdd()`

SPEED

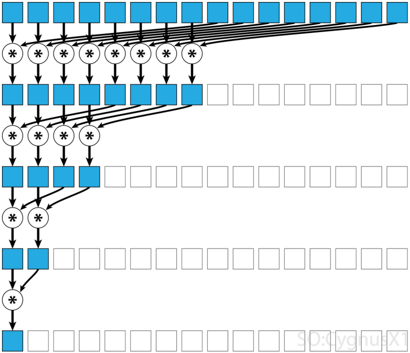
```
broccoli>> ./a.out 10000000 128 3
Kernel 3 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.9872
broccoli>> ./a.out 10000000 128 4
Kernel 4 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.6389 ***
```

Exercise: A True Reduction

```
// Perform a true multi-thread reduction using shared memory
__global__ void array_sum_5(int length, float* data, float *sum)
{
  extern __shared__ float blockvals[];
  blockvals[threadIdx.x] = 0.0;

  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if(idx < length){
    blockvals[threadIdx.x] = data[idx];
  }

  __syncthreads();                // WHY IS THIS NEEDED
                                  // WHAT DOES THIS LOOP DO
  for(int i=blockDim.x/2; i > 0; i /= 2){
    int partner = threadIdx.x + i;
    if(threadIdx.x < i){
      blockvals[threadIdx.x] += blockvals[partner];
    }
    __syncthreads();              // WHY IS THIS NEEDED
  }

  if(threadIdx.x == 0){
    atomicAdd(sum, blockvals[0]);
  }
}
```

Answers: A True Reduction

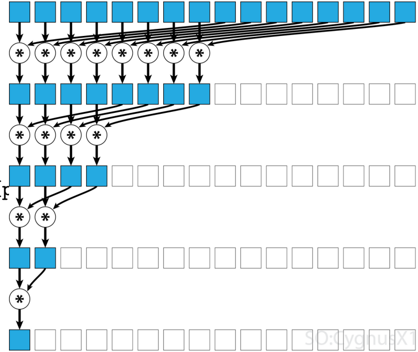
```
for(int i=blockDim.x/2; i > 0; i /= 2){
  int partner = threadIdx.x + i;
  if(threadIdx.x < i){
    blockvals[threadIdx.x] += blockvals[partner];
  }
  __syncthreads();
}
```



Source: WikiOD "CUDA Parallel reduction"

Answers: A True Reduction

```
for(int i=blockDim.x/2; i > 0; i /= 2){
  int partner = threadIdx.x + i;
  if(threadIdx.x < i){
    blockvals[threadIdx.x] += blockvals[p
  }
  __syncthreads();
}
```



Source: WikiOD "CUDA Parallel reduction"

Answers: A True Reduction

- First `syncthreads()` ensures all threads have populated their part of the block-shared array
- Loop performs reduction: each iteration has half remaining threads add on a partner value
- Number of active threads is reduced each time
- MUST `__syncthreads()` after each iteration to ensure adds complete
- Thread 0 ends with final sum and atomically adds

SPEED

See `arraysum-timing.txt` for all times

```
Kernel 3 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.9872
Kernel 4 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.6389 ***
Kernel 5 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.8909
```

Well that was sort of a wasted effort...

Timing in `arraysum_cuda.cu`

- CUDA provides its own timing for GPU-specific events
- Standard `clock()` functions measure CPU while `timeofday()` funcs are in CPU which is running asynchronously from GPU
- Typical timing pattern is

```
cudaEvent_t beg, end;          // timers provided by CUDA
cudaEventCreate(&beg);
cudaEventCreate(&end);

cudaEventRecord(beg);          // start time

// code to measure execution time

cudaEventRecord(end);          // finish time
cudaEventSynchronize(end);     // ensure device / cpu in sync
float gpu_millis = 0;          // calculate elapsed time
cudaEventElapsedTime(&gpu_millis, beg, end);
```

cuBLAS for the Win

- Reduction is tricky to get right and at the point you want to do it, look around for a library
- CUDA provides the cuBLAS with predefined routines for many linear algebra operations (matrix multiply, matrix vector multiply, norms, etc.)
- Example in `arraysum_cublas.cu`

```
cudaEventRecord(beg);
status = cublasSdot(handle,      // dot product routine for floats
                    length,      // length of array to sum
                    dev_x,   1,  // array to sum, step size 1
                    dev_one, 0,  // single 1.0, step size 0
                    dev_sum);    // where to put answewr
cudaEventRecord(end);
```

SPEED

```
Kernel 3 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.9872
Kernel 4 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.6389 ***
Kernel 5 nblocks 78125 nthreads  128 sum: 10000000.0 gpu_millis: 0.8909
cudablasSdot sum:   10000000.0                        gpu_millis: 0.2590 !!!
```

Somebody at NVidia knows their chip well. Stand on their shoulders.

Next Time

- Wrap up CUDA by discussing 2D and 3D indexing
- Possible further discussion of architecture and effects
- Possible discussion of sorting in CUDA

Multi-Dimension Indexing

- Have used single-dimension indexing for most of our discussion so far

  ```
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  ```

- CUDA targets 2D and 3D data types allowing `threadIdx.x`, `threadIdx.y`, `threadIdx.z` to be used
- Kernel must launch with appropriate dimensions via `dim3` data type

```
// hello2D.cu
int thread_x=4, thread_y=2;
int block_x=3, block_y=5;

dim3 threadsPerBlock(thread_x, thread_y);
dim3 blocksPerGrid(block_x, block_y);

hello_gpu2D<<<blocksPerGrid, threadsPerBlock>>>();
```

Example: Matrix-Matrix Addition

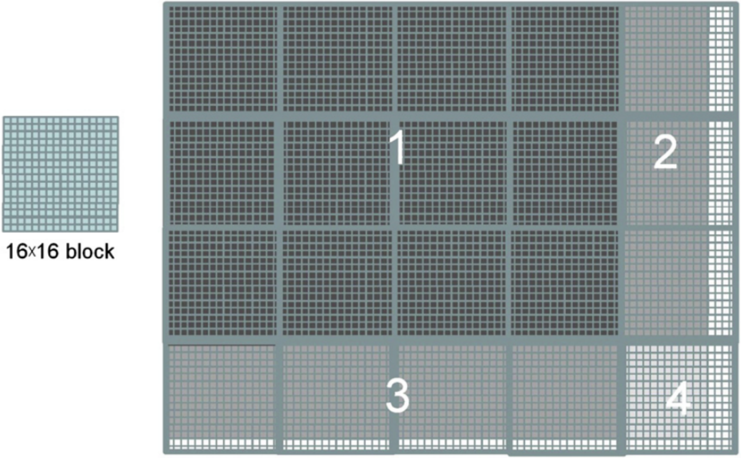


FIGURE 4.5

Covering a 76×62 picture with 16×16 blocks.

CUDA Multi-Dimensional Memory Transfer

To squeeze more performance out, CUDA will pad rows allowing each row to be more efficiently accessed (banked memory)

```
cudaMallocPitch ( void** devPtr, size_t* pitch, size_t width, size_t height )
// allocate 2D array on GPU where each row is padded to be in a
// different memory bank allowing more efficient parallel
// access. `pitch` is set to be the actual width in bytes of a row.

cudaMemcpy2D( void* dst, size_t dpitch, const void* src, size_t spitch,
              size_t width, size_t height, cudaMemcpyKind kind )
// like cudaMemCpy but tailored to 2D arrays w/ width in bytes, height
// in count, and a possible "pitch" for each to indicate padding in
// rows created via cudaMallocPitch().
```

Creates some headaches index calculations later.

Highlights from matadd_cuda.cu

```
////////////////////////////////////////
// memory transfer to device
  float *host_a = (float *) malloc( sizeof(float)*rows*cols );
  float *dev_a;
  cudaMallocPitch((void**) &dev_a, &pitch_a, width, rows);

  cudaMemcpy2D(dev_a, pitch_a, host_a, sizeof(float)*cols,
               sizeof(float)*cols, rows, cudaMemcpyHostToDevice);

////////////////////////////////////////
// kernel launch
  int blockx = (rows + threadx - 1) / threadx;
  int blocky = (cols + thready - 1) / thready;
  dim3 blocks(blockx, blocky);
  dim3 threads(threadx, thready);
  matrix_add<<<blocks, threads>>>(pitch_a, rows, cols, dev_a, dev_b, dev_c);

////////////////////////////////////////
// kernel code
__global__ void matrix_add(long pitch, long rows, long cols,
                           float* a, float* b, float* c)
{
  long row = threadIdx.x + blockDim.x * blockIdx.x; // x : vertical position (row)
  long col = threadIdx.y + blockDim.y * blockIdx.y; // y : horizontal position (col)
  long fpitch = pitch / sizeof(float);              // padded floats per row
  long idx = row * fpitch + col;                    // linear index into matrix
  if(row < rows && col < cols){
    c[idx] = a[idx] + b[idx];
  }
}
```

Exercise: Simple Matrix-Matrix Multiplication

- Formulate matrix multiplication via CUDA
- Perform multiple operations per thread
  - Don't do a single multiple/add per thread
  - Too many threads, too inefficient
- Describe the mapping of work to thread and the total threads required

Answers: Simple Matrix-Matrix Multiplication

- For square $N \times N$ matrix mult, use N^2 threads
- Each thread computes a single output element thus has a row/col index that is unique
- Can compute via a loop

```
// thread i,j runs following loop
float sumij = 0.0;
for(long k=0; k < N; k++){
  sumij += A[i][k] * B[k][j];
}
C[i][j] = sumij;
```

- No locking required

MatMult 1: One Thread Per Output, Diagram

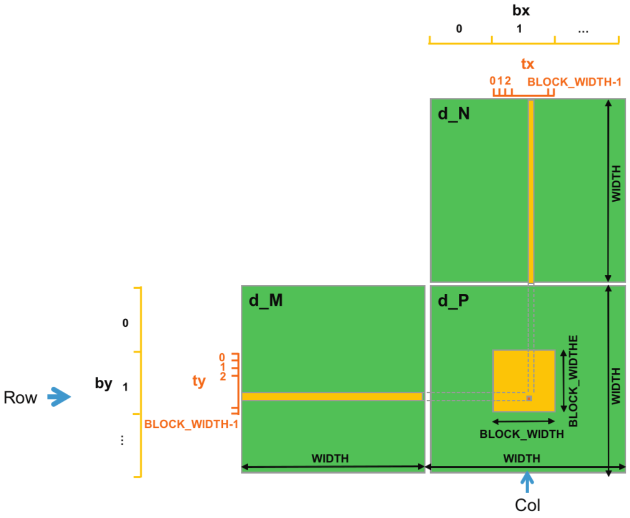


FIGURE 4.6

Matrix multiplication using multiple blocks by tiling d_P.

Source: Programming Massively Parallel Processors by Kirk and Hwu

Exercise: Strategies to Improve Performance

- The previous method is limited somewhat in performance
- Identify a bottleneck with the below and how one might solve it

```
// thread i,j runs following loop
float sumij = 0.0;
for(long k=0; k < N; k++){
  sumij += A[i][k] * B[k][j];
}
C[i][j] = sumij;
```

Hint: how did we improve performance in previous kernels?

Answers: Strategies to Improve Performance

- Repeated main memory accesses slow down basic kernel
- Must exploit cache to get better performance
- Thread Block loads a chunk of the matrix and shares it
- Referred to as a "tiled" matrix approach in several spots
- Requires mild reformulating of matrix multiply as block/tiled operations

MatMult Tiled Diagram

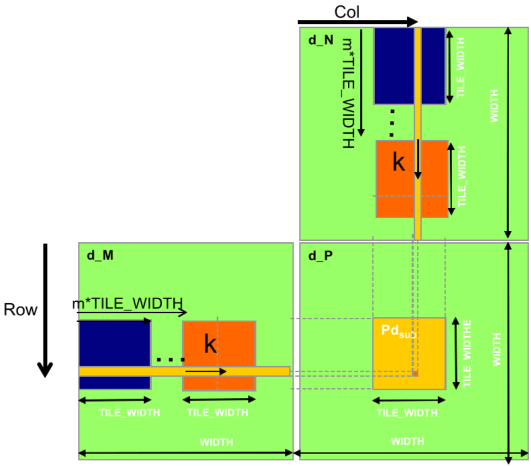


FIGURE 5.13

Calculation of the matrix indices in tiled multiplication.

Source: Programming Massively Parallel Processors by Kirk and Hwu

Sorting on GPUs

- As promised, briefly revisit sorting on GPUs
- Note landscape is a bit different
  - Cores much less than elements on Distributed / Shared Systems
  - Many Threads/Cores available on GPUs
  - (More) Viable to consider $N = P$
- Worth reconsidering some algorithms which were skipped as impractical previously
- Stat with Odd-Even Sort

Exercise: Odd-Even Sort Revisited

- Variant of bubble sort which splits bubbling into odd/even phases
- $O(N^2)$ complexity of serial algorithm
- There is potential for parallelism here: what is it?
  - Consider simple case where each $P = N$: each proc hold a single number
  - What can be parallelized and how?

```
ODD_EVEN_SORT(A[]) {
  N = length(A[])
  for(r=0 to N-1){
    if(r is even){
      for(i=0; i<N-1; i+=2){
        compare_exchange(A, i, i+1);
      }
    }
    if(r is odd){
      for(i=1; i<N-1; i+=2){
        compare_exchange(A, i, i+1);
      }
    }
  }
}

COMPARE_EXCHANGE(A[], i, j){
  if(A[i] > A[j]){
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
  }
}
```

Answers: Odd-Even Sort

- There is potential for parallelism here: what is it?
- Consider simple case where each $P = N$: each proc hold a single number
- What can be parallelized and how?
  - *The inner loops of `compare_exchange()` can be executed in parallel as it involves communication between 2 procs to potentially exchange elements but only with a single partner.*
  - *Even iterations, lower evens exchange with higher odds*
  - *Odd iterations lower odds exchange with higher evens*
  - *Single CUDA Threads can perform compare/exchange on global array elements*

Odd-Even Sort CUDA Code

```
// oddeven_cuda.cu
__global__ void odd_even_round(float *data, int length)
{
  int idx = 2 * (threadIdx.x + blockDim.x * blockIdx.x);
  if(idx < length-1){
    float x = data[idx+0];
    float y = data[idx+1];
    float newx = min(x,y);
    float newy = max(x,y);
    data[idx+0] = newx;
    data[idx+1] = newy;
  }
}

int main(){
  ...;
  for(int i=0; i<length; i++){
    if(i % 2 == 0){
      odd_even_round<<<nblocks, nthreads>>>(dev_x, length);
    }
    else{
      odd_even_round<<<nblocks, nthreads>>>(dev_x+1, length-1);
    }
  }
  ...;
```

Complexity Analysis + Performance

- Assuming
  - $O(N)$ procs ($N/2$ threads)
  - N Steps
- $O(N)$ time complexity in theory but...
- Overhead kills practical efficiency

```
>> nvcc oddeven_cuda.cu
>> ./a.out 500000 128
length      500000 nblocks   1954 nthreads  128
gpu_millis:  3195.8342
cpu_millis:    94.7070   # libc's qsort()
```

  - Kernel launches required for sync across blocks
  - No use of cached memory

Improvements on Odd-Even Sort

Compare-Split on Array Chunks

- Rather than single elements, work array-chunks
- Thread blocks
  - Load two array chunks to shared cache
  - Threads sort combined chunks (in parallel?)
  - Write low/high chunks back to memory

Bitonic Sort and Batcher's Odd-Even Sort

- Odd-even does `compare_swap(a[i], a[i+1])` in all N iterations
- Sorting networks vary this each iteration `compare_swap(a[i], a[i+8])`
- Correct sequences of comparisons yields $O(\log^2 N)$ iterations with N procs while preserving correctness
- Targeted at hardware with fixed input sizes (e.g 16 inputs) but applicable particularly to sorting within a Thread Block

GPU Sorting is an Active Research Topic

1032 Int J Parallel Prog (2018) 46:1017–1034

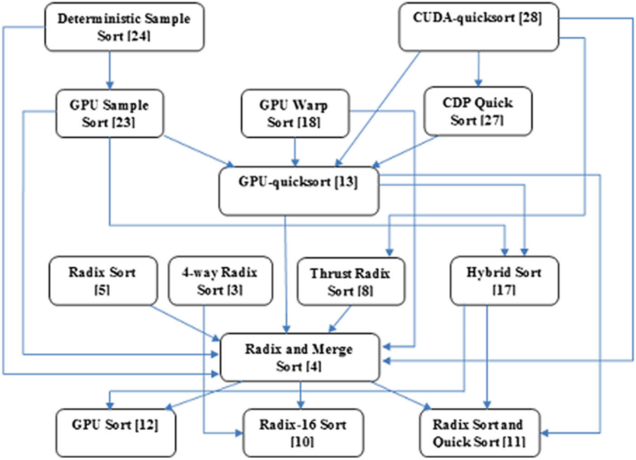


Fig. 1 Performance comparison of the algorithms

Source: Survey of GPU Based Sorting Algorithms by Singh et al in International journal of parallel programming, 2017. (UMN Library) (DOI Link)

CUDA Alternatives

OpenCL

- "Open source" "version" of CUDA
- Similar in nature: program `__kernel__` functions, explicitly manage memory
- Supports multiple devices including AMD/ATI graphics cards, NVidia Cards, Intel Graphics, Apple Graphics
- Performance can usually match CUDA with enough hand-tuning

OpenACC

- Like OpenMP: directive based parallelism for GPU
- Specify accelerator execution via `#pragma acc`
- Supports "accelerator" devices like GPUs without need to define kernels
- Support in some compilers like GCC