Accelerated Systems
046278
EE - Technion

Tutorial:
Intermediate & Advanced CUDA

Amir Watad
Agenda

• Basic Parallel Patterns
• Task Level Parallelism
• Optimizations
Basic Parallel Patterns

• Reduce

• Scan
Reduce

- Inputs:
  - Array of items of type T.
  - Commutative & Associative binary operator:
    - $T = T \text{ OP } T$

- Output:
  - A single item of type T.
Examples

- Sum of array
- Minimum of array
- Dot product
\[ A + E + C + G \]

\[ B + F + D + H \]
Reduce Kernel: 1st Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        half_length /= 2;
    }
}
```
Reduce Kernel: 1st Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        half_length /= 2;
    }
}
```

Problems?
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        half_length /= 2;
        __syncthreads();
    }
}
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        half_length /= 2;
        __syncthreads();
    }
}
Reduce Kernel: 3rd Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        if (tid < half_length) {
            half_length /= 2;
            __syncthreads();
        }
    }
}
```
Reduce Kernel: 3rd Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        if (tid < half_length) {
            half_length /= 2;
            __syncthreads();
        }
    }
}
```
Reduce Kernel: 3rd Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        if (tid < half_length) {
        }
        half_length /= 2;
        __syncthreads();
    }
}
```

Idle threads are still there!
Reduce Kernel: 3rd Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        if (tid < half_length) {
        }
        half_length /= 2;
        __syncthreads();
    }
}
```

Any problems now?
Reduce Kernel: 4th Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
    }
    __syncthreads();
    half_length /= 2;
}
```

Not always necessary. Depends on #threads.
Reduce Kernel: 4th Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
}
```

Ex. for the reader:
What if length is not a power of 2?
Reduce Kernel: 4th Try

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
}
```

There is room for optimizations. We’ll come back to it later.
Multiple threadblocks

• Step 1: Each threadblock does reduction on part of the array.

• Step 2: Now we have a small array of results. Use one threadblock for final reduce.
Step 1

```c
__global__ void reduce_first(float *A, int length, float *out) {
    int tid = threadIdx.x;
    int length_per_block = length / blockDim.x;
    int start = blockIdx.x * length_per_block;
    int half_length = length_per_block / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
    if (tid == 0) {
        out[blockIdx.x] = A[start];
    }
}
```
Step 2

- Simple reduction on out array
#define LEN 4096

int main() {
    float *A, *gpu_A, *gpu_out;
    A = malloc(LEN * sizeof(float));

    /* fill A with numbers ... */

    int nblocks = LEN / 2048; /* 1024 threads are good for 2048 items */
    cudaMalloc(&gpu_A, LEN * sizeof(float));
    cudaMemcpy(gpu_A, A, LEN * sizeof(float), cudaMemcpyHostToDevice);
    cudaMalloc(&gpu_out, nblocks);

    reduce_first<<<nblocks, 1024>>>(gpu_A, LEN, gpu_out);
    cudaDeviceSynchronize();
    reduce<<<1, nblocks/2>>>(gpu_out, nblocks);
    cudaDeviceSynchronize();

    float result;
    cudaMemcpy(&result, gpu_out, sizeof(float), cudaMemcpyDeviceToHost);
    printf("result %f\n", result);
    return 0;
}
# Prefix Sum / Scan

<table>
<thead>
<tr>
<th>Inclusive Scan (+)</th>
<th>0</th>
<th>A</th>
<th>A + B</th>
<th>A + ... + C</th>
<th>A + ... + D</th>
<th>A + ... + E</th>
<th>A + ... + F</th>
<th>A + ... + G</th>
<th>A + ... + H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive Scan (+)</td>
<td>A</td>
<td>A</td>
<td>A + B</td>
<td>A + ... + C</td>
<td>A + ... + D</td>
<td>A + ... + E</td>
<td>A + ... + F</td>
<td>A + ... + G</td>
<td></td>
</tr>
</tbody>
</table>
How to parallelize

- Kogge-Stone Algorithm

(Programming Massively Parallel Processors, 3rd Edition)
__global__ void kogge_stone_scan(float *A, int length) {
    int tid = threadIdx.x;
    int increment;
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if (tid >= stride) {
            increment = A[tid - stride];
        }
        __syncthreads();
        if (tid >= stride) {
            A[tid] += increment;
        }
        __syncthreads();
    }
}
Work Efficiency

• In serial Algorithms, we care about execution time.

• In parallel Algorithms, we also care about execution time.

  • But we also care about work efficiency. (Why?)

  • Total work done by all threads together.
Kogge-Stone’s Work Efficiency

• Speed = Execution time = O(log(N))

• Work Efficiency: O(N * log(N))

• Serial Scan: O(?)

• More efficient algorithms for scan exist
  • Although more complicated
  • We’ll skip them for now.
Histograms

- Only if we have time.
- Otherwise, will be explained in the homework assignment.
Agenda

• Basic Parallel Patterns
• Task Level Parallelism
• Optimizations
Task Level Parallelism

• Until now we talked about Data parallelism: Parallelizing a single task.
• What if we have multiple tasks?
  • What if in addition some of them are dependent?
Example

- Input: 64 vectors
- Output: Minimum of each vector.
Solution 1

```c
int main() {
    vector_t *vectors[64];
    float *gpu_vectors[64];
    int f[100]; /* files */

    /* initialize f ... */
    /* allocate gpu_vectors[i] */

    for (int i = 0; i < 64; i++) {
        load_vector_from_file(f[i], vectors[i], 0);
    }

    for (int i = 0; i < 64; i++) {
        cudaMemcpy(gpu_vectors[i], vectors[i].data,
                    vectors[i].length * sizeof(float), cudaMemcpyHostToDevice);
        reduce<<<1, vectors[i].length / 2>>>(gpu_vectors[i], vectors[i].length);
        cudaMemcpy(vectors[i].sum, gpu_vectors[i],
                    sizeof(float), cudaMemcpyDeviceToHost);
    }
    cudaMemcpy(vectors[i].sum, gpu_vectors[i],
                sizeof(float), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    return 0;
}
```

Problems?
int main() {
    vector_t *vectors[64];
    float *gpu_vectors[64];
    int f[100]; /* files */

    /* initialize f ... */
    /* allocate gpu_vectors[i] */

    for (int i = 0; i < 64; i++) {
        load_vector_from_file(f[i], vectors[i], 0);
    }

    cudaMemcpy(gpu_vectors[i], vectors[i].data,
                vectors[i].length * sizeof(float), cudaMemcpyHostToDevice);
    reduce<<<1, vectors[i].length / 2>>>(gpu_vectors[i], vectors[i].length);
    cudaMemcpy(vectors[i].sum, gpu_vectors[i],
                sizeof(float), cudaMemcpyDeviceToHost);
}

cudaDeviceSynchronize();
return 0;
}
Synchronous Calls

- Kernel Invocations and Memory movements are synchronous.
- CUDA will serialize them!
GPU Occupancy

- Let’s assume a GPU with 10 SMs.
- SM can run 2048 threads concurrently.
- Average vector length is 1024 items
  - i.e. needs 512 threads for reduce
- We run 512 threads at a given moment
- Out of possible total of 20480.
- 2.5% Occupancy => 97.5% of the GPU is wasted!
It’s even worse!

cudaMemcpy(...., cudaMemcpyHostToDevice);
reduce<<<<<1, vectors[i].length / 2>>>(....);
cudaMemcpy(...., cudaMemcpyDeviceToHost);
Solution 2: Batching

• Idea: Load all vectors into one large vector.

• Single kernel, threadblock per vector.
Batching is Good, But ..

- More difficult if vectors are not the same length
- What if we don’t have all vectors at the same time? (e.g. we implement a network server)
- What if not all tasks are the same?
Comes CUDA Streams

• A stream is a virtual queue.
  • Mapped to hardware queues, called Hyper-Q
• Tasks in the same queue are executed serially.
• Tasks in different queues might be executed in parallel
  • But not guaranteed. Why?
Back to our example

```c
int main() {
    /* initialize ... */

    for (int i = 0; i < 64; i++) {
        cudaMemcpy(gpu_vectors[i], vectors[i].data,
                    vectors[i].length * sizeof(float), cudaMemcpyHostToDevice);
        reduce<<<1, vectors[i].length / 2>>>(gpu_vectors[i], vectors[i].length);
        cudaMemcpy(vectors[i].sum, gpu_vectors[i],
                    sizeof(float), cudaMemcpyDeviceToHost);
    }
    cudaDeviceSynchronize();
    return 0;
}
```

- What tasks are dependent?
Using Streams

```c
int main() {
    /* initialization ... */

    cudaStream_t streams[64];
    for (int i = 0; i < 64; i++) {
        cudaStreamCreate(&streams[i]);
    }

    for (int i = 0; i < 64; i++) {
        cudaMemcpyAsync(gpu_vectors[i], vectors[i].data,
                         vectors[i].length * sizeof(float),
                         cudaMemcpyHostToDevice, streams[i]);
        reduce<<<1, vectors[i].length / 2, 0, streams[i]>>>(...);
        cudaMemcpyAsync(vectors[i].sum, gpu_vectors[i],
                         sizeof(float),
                         cudaMemcpyDeviceToHost, streams[i]);
    }
    cudaDeviceSyncronize();
    return 0;
}
```
What streams give us

• Specify (lack of) dependencies between tasks.

• Overlap data with compute.
Cleanup

• In real life, you’ll want to cleanup after yourself

• cudaStreamDestroy(stream)

• And don’t forget about checking CUDA calls for errors. Always!
More Stream related

- `cudaStreamQuery(stream)`
  - Checks if all submitted tasks to this stream are done.

- `cudaStreamSynchronize(stream)`
  - Block until all tasks in stream are done
Streams and Memory

- cudaMemcpyAsync between CPU and GPU expects CPU memory to be pinned.

- Why?

- Will work with non-pinned memory. But will be serialized! Be careful.

- For pinned memory allocation use:
  - cudaMemcpyAsync(&memptr, size, 0)
Measuring time

- We want to measure how much time a computation takes (including memory movements).

- With synchronous CUDA calls it’s easy:
  - Measure time on CPU before call and after `cudaDeviceSynchronize()`

- Not so with streams. (Why?)
CUDA Events

- Think about an event as a NOP task.
- Does not do “useful” work.
- But records the time of its “execution”.
Example

- Compute the average time it takes for a reduce kernel to execute. Excluding memory movements.
for (int i = 0; i < 64; i++) {
    cudaMemcpyAsync(gpu_vectors[i], vectors[i].data,
    vectors[i].length * sizeof(float),
    cudaMemcpyHostToDevice, streams[i]);
    reduce<<<1, vectors[i].length / 2, 0, streams[i]>>>(...);
    cudaMemcpyAsync(vectors[i].sum, gpu_vectors[i],
    sizeof(float),
    cudaMemcpyDeviceToHost, streams[i]);
}
Measuring time with events

cudaEvent_t events_start[64];
cudaEvent_t events_stop[64];

for (int i = 0; i < 64; i++) {
    cudaEventCreate(&events_start[i], 0);
    cudaEventCreate(&events_stop[i], 0);
}

for (int i = 0; i < 64; i++) {
    cudaMemcpyAsync(...);
    cudaEventRecord(events_start[i], streams[i]);
    reduce<<<1, vectors[i].length / 2, 0, streams[i]>>>(gpu_vectors[i], vectors[i].length);
    cudaEventRecord(events_stop[i], streams[i]);
    cudaMemcpyAsync(...);
}
cudaDeviceSyncronize();

float total_time_in_ms = 0;
for (int i = 0; i < 64; i++) {
    float time;
    cudaEventElapsedTime(&time, events_start[i], events_stop[i]);
    total_time_in_ms += time;
}
float avg_time = total_time_in_ms / 64;
Task parallelism with events

• Streams are good for linear dependencies.

• However, they are not good for more complex patterns.
Task parallelism with events

- We use events for such cases
Task parallelism with events

A <<<..., stream[0] >>>();
B <<<..., stream[0] >>>();
C <<<..., stream[1] >>>();
D <<<..., stream[1] >>>();
E <<<..., stream[0] >>>();
Task parallelism with events

A Start.
But something is missing

A<<<..., stream[0]>>>();
B<<<..., stream[0]>>>();
C<<<..., stream[1]>>>();
D<<<..., stream[1]>>>();
E<<<..., stream[0]>>>();
Task parallelism with events

If we could only force E to wait for D

A<<<..., stream[0]>>>(());
B<<<..., stream[0]>>>(());
C<<<..., stream[1]>>>(());
D<<<..., stream[1]>>>(());
E<<<..., stream[0]>>>(());
Task parallelism with events

If we could only force E to wait for D

cudaEvent_t event;
cudaEventCreate(&event, cudaEventDisableTiming);
A<<<..., stream[0]>>>();
B<<<..., stream[0]>>>();
C<<<..., stream[1]>>>();
D<<<..., stream[1]>>>();
cudaEventRecord(event, stream[1]);
cudaStreamWaitEvent(stream[0], event, 0);
E<<<..., stream[0]>>>();
More events

- `cudaEventQuery` - check whether event was triggered.

- Events are also good for synchronizing streams across GPUs. (A stream cannot have tasks on different GPUs)
When events don’t help

- Streams and Events assume that we know the execution plan beforehand.
- Not always true.
  - We might run different computations based on the result of previous computation.
  - Maybe similar computations, but #threads or #blocks change based on results
When events don’t help

A -> B
C -> D
D' -> E

If result == X
If result == Y
When events don’t help

Actually we can use events if we really want to:
- Execute both D and D’.
- Make each look at the previous result, and terminate if not meant for it
- Breaks modularity.
- Wastes resources. Think tens of possible combinations in each step.
So what to do in such cases?
A<<<..., stream[0]>>>();
B<<<..., stream[0]>>>();
cudaStreamWaitEvent(stream[0], eventD, 0);
E<<<..., stream[0]>>>();
C<<<..., stream[1]>>>();
cudaEventRecord(eventC, stream[1]);

while (cudaEventQuery(eventC) == cudaErrorNotReady);
if (result == X) {
    D<<<..., stream[1]>>>();
} else {
    D'<<<..., stream[1]>>>();
}
cudaEventRecord(eventD, stream[1]);
A<<<..., stream[0]>>>();
B<<<..., stream[0]>>>();
cudaStreamWaitEvent(stream[0], eventD, 0);
E<<<..., stream[0]>>>();
C<<<..., stream[1]>>>();
cudaEventRecord(eventC, stream[1]);

while (cudaEventQuery(eventC) == cudaErrorNotReady) {
    if (result == X) {
        D<<<..., stream[1]>>>();
    } else {
        D'<<<..., stream[1]>>>();
    }
    cudaEventRecord(eventD, stream[1]);
}

 Might become complicated in bigger task graphs.

 Might require multi-threads in the host if waiting for several unrelated events.
Agenda

• Basic Parallel Patterns
• Task Level Parallelism
• Optimizations
Optimizations

• Let’s optimize our code!
“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming”
- Donald Knuth
Remember

• When you optimize you compromise.

• No free lunch (usually).
  • More complicated code. Less sleep. No life.

• Think how innocent you were before you knew CUDA.

• It’s a dark path.
Think first!

- Does performance really matter (in my case)?
  - Not always a binary question.
  - Deliver slower product today or faster product next year?
- What performance am I expecting?
- Is the problem in my code?
  - The machine is slow? Wrong compilation flags?
- What are the program’s bottlenecks?
Set your goals

• What performance do you want to have.

• Is it realistic?

• What’s the performance of your competition.
Use intuition

• Is there something terribly wrong?

• e.g. if 100 items vector sum takes 10 seconds, something is terribly wrong.

• What could it be?
  • Another program is using the GPU?
  • The GPU is too hot?
  • You measure the time wrong? Using debug compilation flags?
  • You’re running the wrong code?
  • You have HDD instead of SSD? Weak GPU instead of what you think you have? (know your machine!)
Compilation flags

• You debug with: -g -G

• You execute with: -O3
Ok, your code is the problem

- But which part?
- Profile first. Never go blind!
- gettimeofday(), CUDA Events, clock64()
- Use NVIDIA’s visual profiler.
Nvidia’s Profiler
Profiling has overheads

- Make sure your times aren’t affected too much by the measurements!
The bottleneck

• Now you have the times.

• You can either be happy or sad.
Can you fix it?

- External library? Is there a better one?
- Bad algorithm? Can be replaced with better one?
  - A good algorithm on CPU isn’t always good on GPU.
Levels of optimization

- Algorithm and Data structures
- Algorithm / Data structure tuning
  - e.g. is there a price you pay for checking for corner cases? Can you remove corner cases?
- Code tuning: Are you using doubles where you could use floats? Cartesian instead of polar?
- More aggressive code tuning: Machine dependent, assembly level.

Adapted from: Programming Pearls (2nd Edition)
With GPUs we have another level

• Are we fully occupying the GPU?

• Start here if your problem is in GPU code.

• Also memory access patterns
  • Can be thought of as part of the data structure
Start from the higher levels

- Almost always easier
- Almost always gives better performance gains
Occupancy

• % of total available threads that we are using.

• Can be limited by:

  • #Registers per thread.
  • #Threadblocks
  • shared memory
  • Not enough parallelism.
Occupancy

• Higher is better

• In general. But not always

• 2048 in each SM share the same ALUs.

• So if we have (e.g.) 120 ALUs, why have 2048 threads?
  • Latency hiding.
Algorithms

- Think about speed and work efficiency.
- They might be both important, depending on your code.
- Load balance: Do all threads have work all the time?

- Some algorithms are efficient for serial code, but cannot be parallelized efficiently. Choose wisely.

- $O$ notation is only asymptotic
  - $O(N \times \log(N))$ can be better than $O(N)$ for your inputs
  - e.g. $N \leq 1000$, linear coefficient 100.
Reduce Kernel

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
}
```

Some threads are idle most of the time.
- Maybe use less threads?
- Or break into 2 or more kernels with different number of threads?
- Not always good. There are overheads for kernel invocation.
Data Access

• Memory access matters in GPUs.
• Which memory you access.
• How you access it.
Global Memory - Coalescing

- GPU memory accessed in **cache line** granularity.
- Make your threads (in each warp) concurrently access as fewer cache lines as possible.
- Access bytes [0..127] = Good. (1 cache line)
- Access bytes [1..128] = Less good (2 cache lines)
Shared Memory

• Are you accessing same data multiple times?
• Load it to the shared memory. It’s much faster.

• Global memory latency: ~500ns
• Shared memory latency: ~5ns
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
}
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length/ 2;
    __shared__ float sharedA[1024];
    for (int i = tid; i < length; i += blockDim.x) {
        sharedA[i] = A[i];
    }
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
    if (tid == 0) {
        A[0] = sharedA[0];
    }
}
**Reduce Kernel**

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length / 2;

    __shared__ float sharedA[1024];
    for (int i = tid; i < length; i += blockDim.x) {
        sharedA[i] = A[i];
    }
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
    if (tid == 0) {
        A[0] = sharedA[0];
    }
}
```

If size is known at compile time
Reduce Kernel

If size is known only at invocation time

```c
__global__ void reduce(float *A, int length) {
    int tid = threadIdx.x;
    int half_length = length/ 2;
    extern __shared__ float sharedA[];
    for (int i = tid; i < length; i += blockDim.x) {
        sharedA[i] = A[i];
    }
    while (half_length >= 1) {
        for (int i = tid; i < half_length; i += blockDim.x) {
        }
        __syncthreads();
        half_length /= 2;
    }
    if (tid == 0) {
        A[0] = sharedA[0];
    }
}

int main() {
    reduce<<<1, nthreads, sharedMemorySizeBytes>>>(A, length);
}
```
Compute Intensity

- Compute Intensity: Compute / Data Items
- Matrix multiplication: $N^3$ compute, $N^2$ items
  - $\Rightarrow$ each memory item is reused $N$ times.
- Intensity $> 1 \Rightarrow$ Potential for gain using shared memory.
Regularization

• Example:
  • Matrix in global memory is in row major format.
  • We want to do computations column wise.
  • Not efficient.

• Solution:
  • read the matrix in a coalesced way, and write it transposed to shared memory.
Global Memory

Shared Memory

Load and Transpose
No free lunch though: Shared memory bank conflicts

- Shared memory consists of 32 Banks of width 4B.
- Item i goes to bank $i \mod 32$
- If 2 different threads concurrently access different items on the same bank == Bank conflict.
  - Bank conflict = Serialized memory accesses.
  - No problem if they access same item (broadcast)
- Be careful with random / strided / irregular access to shared memory.
Shared memory lifetime and scope

- Accessible within the threadblock only
- Evaporates once kernel terminates
Thread Divergence

- Threads are grouped into warps
- All threads in a warp run at lock-step
- If they want to do different tasks at the same time, they wait for each other => serialization
int x;

if (threadIdx.x == 0) x = 1;  /* threads 1 - 31 wait */
else                      x = 2;  /* thread 0 waits */
Could be worse

switch (threadIdx.x % 32) {
    case 0: ...; break;
    case 1: ...; break;
    ...
    case 31: ...; break;
}

Intrinsics

- CUDA has fast built-in math functions.
- Again, no free lunch: They are approximations only.
- But that might be good enough for your application.
- __cosf (instead of cosf), __expf (instead of expf).
- f is for float: __expf is faster than expf which is faster than exp (double).