KU LEUVEN

Geavanceerde Computerarchitectuur
5^{th} Session

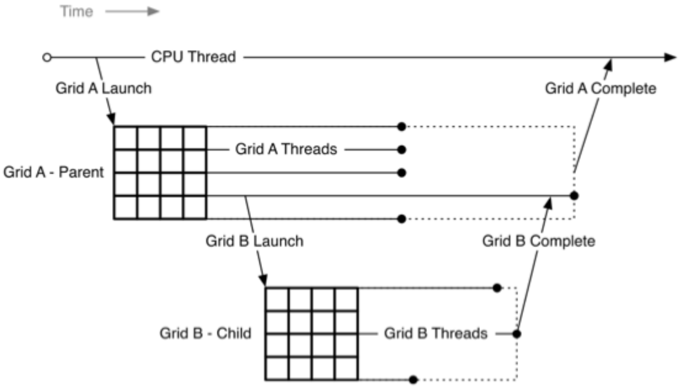
Dynamic parallelism and asynchronous operation

Alex Marinsek, alexander.marinsek@kueluven.be
DRAMCO – WaveCore – ESAT
Academic year 2023/24

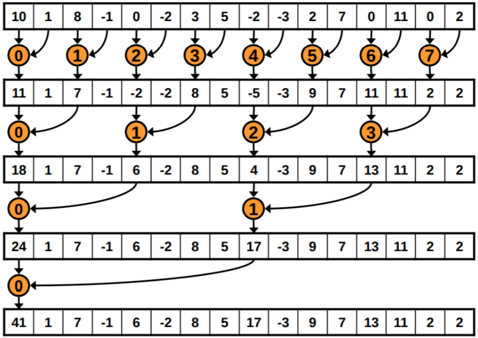
0 Outline

1 Outline

1 Spawn kernel from within kernel

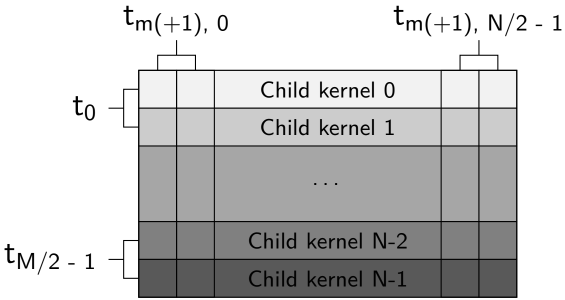


1 Example – Recall reduction and the main bottleneck

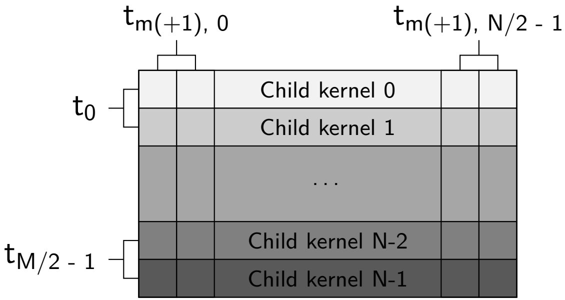


- Had to synchronize threads on each reduction step
- Could only synchronize within a block (1024 threads)
- Could handle larger data $(N > 2048)$ by striding, calling the kernel multiple times, or by using multiple blocks and `atomicMax()`, among others

1 Example – dynamic parallelism for 2D array reduction



1 Example – dynamic parallelism for 2D array reduction



Requires at least compute capability 3.5 (lab PCs have 3.0)

2 Outline

1. Dynamic parallelism
2. Asynchronous operation
3. Exercise

2 Offload to GPU (B) without blocking the CPU (A)

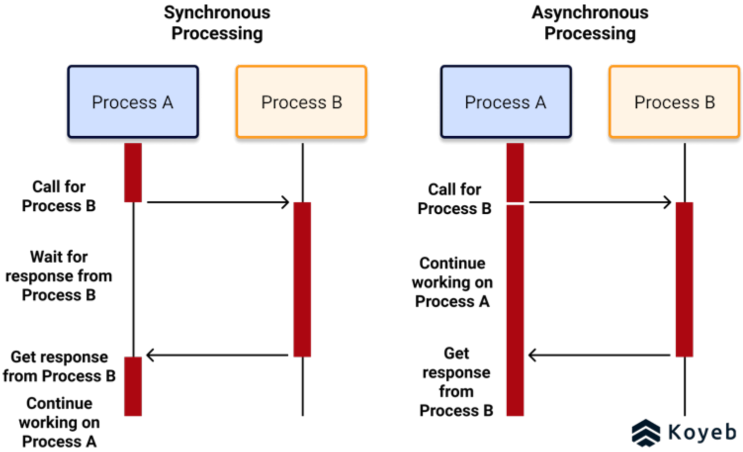


Image from Koyeb

2 (A)synchronous memory copying

```
// Copy memory synchronously (wait/block till finished)
cudaMemcpy ( dst, src, count, kind );
...
// Copy memory asynchronously (no waiting, continue with CPU code)
cudaMemcpyAsync ( dst, src, count, kind, stream );
```

A handy link to the CUDA API reference

2 Kernel invocation is asynchronous

```
// This is an asynchronous call by default
foo<<<blk, thr>>>(...);
...
// Can synchronize manually (wait till GPU finished all of its tasks)
cudaDeviceSynchronize();
```

2 Kernel invocation is asynchronous

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);

foo<<<blk, thr>>>(...);

cudaEventRecord(stop);

// This also waits
cudaEventSynchronize ( stop );
```

Think of a stack (bottom up): `start`, `foo`, `stop`

2 The "GPU call stack" is ordered

Copy memory to GPU, execute, and copy back to CPU in this order and without blocking the CPU

```
// To GPU
cudaMemcpyAsync ( dst , src , count , kind , stream );

// Execute
foo<<<blk , thr>>>(...);

// Back to CPU
cudaMemcpyAsync ( dst , src , count , kind , stream );
```

2 CUDA streams

There can be more than one such "call stack", called a stream to (example):

- Distinguish between GPU calls (a call for each incoming request/user)
- Prioritize certain processing (video rendering vs. video statistics)

```
// Last argument is the stream
cudaMemcpyAsync ( dst, src, count, kind, stream );

// Last argument, again is the stream
foo<<<blk, thr, shared_size, stream>>>(...);

// Wait for the stream to complete
cudaStreamSynchronize( stream )
```

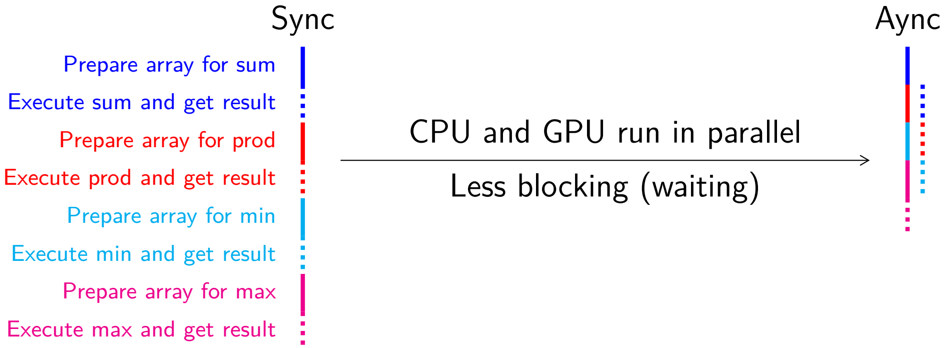
3 Outline

3 Sync vs async execution comparison

We will revisit reduction:

- Perform `sum`, `prod`, `min`, and `max`
- Process 4 independent arrays
- Use a for loop to generate each of the arrays
- Approach 1: do all of this in sequence, synchronously
- Approach 2: do array generation while the previous data is being asynchrounously on the GPU
- Compare the total execution time (including CPU code)

3 The exercise in figure form



3 Pointers

- You already have the reduction kernel from exercise 2
- Implement summation, multiplication, and minimum detection
- Can use the same kernel and change only the computation part see below (or point to a different `__device__` function each time)

```
__global__ foo(...){
    ...
    #if KERNEL==0
    #   val = ... + ...
    #else
    #   val = ... * ...
    #endif
}
```

3 Pointers continued

- Implement a workflow where you define and populate an array on the CPU, before migrating it to the GPU and executing the kernel function.
- Do this 4 times, once for each operation: sum, prod, min, and max.
- Perhaps make 2 CPU functions - one which calls the kernel(s) synchronously, one after the other, and one that calls them while preparing the data for the next kernel.
- Call the two functions from `main()` and get the execution time using the Chrono library (see cheatsheet).
- Make sure to synchronize before stopping the timer.