B3CC: Concurrency

09: GPGPU

Ivo Gabe de Wolff

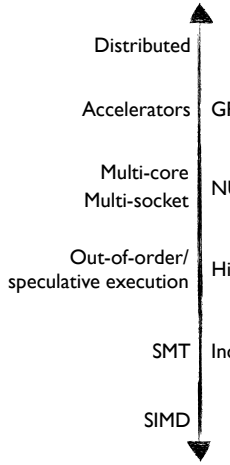
Announcement

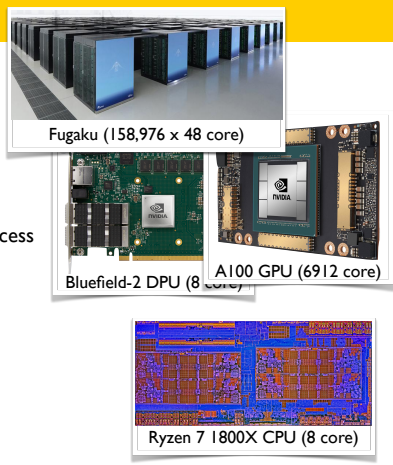
- Mid-term exam next week
  - Tuesday 19-12-2023 @ 13:00 – 15:00 in Olympos Hal 2
  - Covers all the material up to and including STM
  - Excluding Delta-stepping

Announcement

- "Minder massaal" exam
  - Only for students with permission
  - Ruppert D, 13:00 - 15:00
  - Room and time changed!

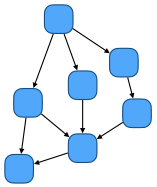
Recap





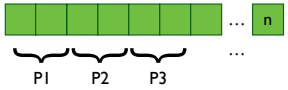
https://en.wikichip.org/wiki/amd/ryzen_7/1800x

Recap



Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM
- Modest parallelism
- Hard to program



Data parallelism

- Operate simultaneously on bulk data
- Implicit synchronisation
- Massive parallelism
- Easy to program

Data parallelism

- Despite the name, data parallelism is only a programming model
  - The key is a *single logical thread of control*
  - It does not actually require the operations to be executed in parallel!
  - Today: let's look at how you would actually implement data-parallel operations, in parallel, on the GPU

CPU vs. GPU

- Traditional CPU designs optimise for single-threaded performance
  - Branch prediction, out-of-order execution, large caches, etc.
  - Much of the available die area is dedicated to *non-computation* resources
  - CPUs are designed to optimise *latency* of an individual thread's results
  - Must be good at everything, parallel or not

https://www.anandtech.com/show/16261/investigating-performance-of-multithreading-on-zen-3-and-amd-ryzen-5000

CPU vs. GPU

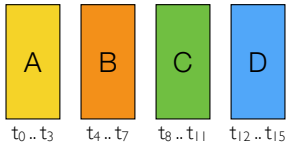
- GPUs are designed to accelerate graphics processing (rasterisation)
  - This is an inherently *data-parallel* task
  - GPUs are designed to maximise *bandwidth*: the time to process as single pixel is less important than the number of pixels processed per second
  - Specialised for compute intensive, highly parallel computation

CPU vs. GPU

- CPU
  - Multiple tasks = multiple threads
  - Tasks run different instructions
  - 10s of complex threads execute on a few cores
  - Threads managed explicitly
  - Expensive to create & manage threads
- GPU
  - SIMD: single instruction, multiple data
  - 10s of thousands of lightweight threads
  - Threads are managed and scheduled by the hardware
  - Cheap to create many threads

CPU vs. GPU

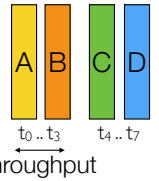
- Image we need to perform some operation that takes 4 units of time (clock cycles), on values A, B, C and D.



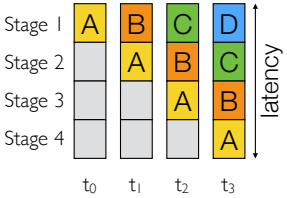
https://en.wikipedia.org/wiki/IP_over_Avian_Carriers

CPU vs. GPU

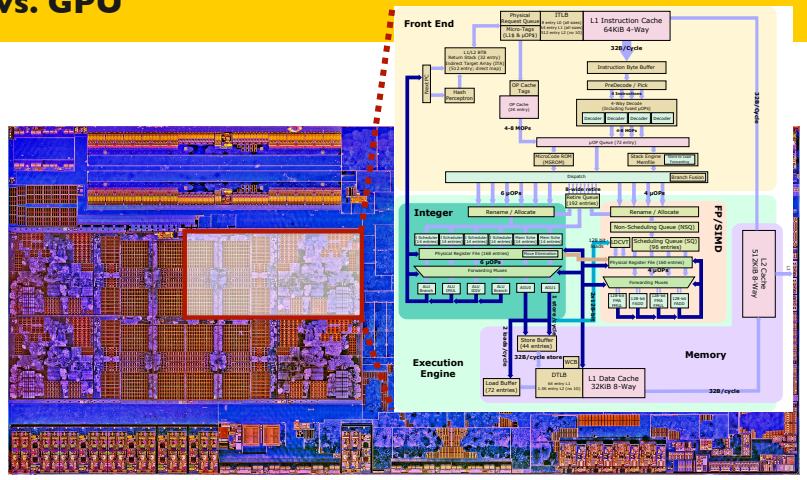
- Horizontal parallelism: increase throughput
  - More execution units working in parallel



- Vertical parallelism: hide latency
  - Keep functional units busy when waiting for dependencies, memory, etc.

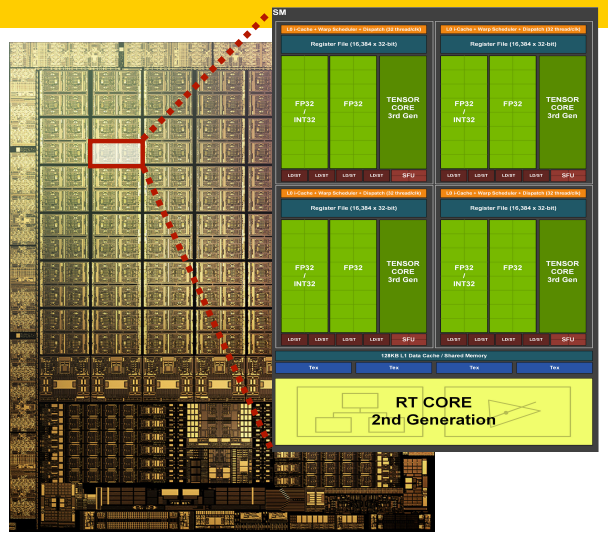


CPU vs. GPU



https://en.wikichip.org/wiki/amd/microarchitectures/zen

CPU vs. GPU



NVIDIA GA102

GPU architecture

- The CPU spends a lot of resources to *avoid* latency
- The GPU instead uses parallelism to *hide* latency
  - No branch prediction
  - One task (kernel) at a time
  - No context switching
  - Limited super-scalar pipeline
  - No out-of-order execution
  - Very low clock speed

GPU architecture

- Each GPU has…
  - A number of *streaming multiprocessors* (comparable to CPU cores)
  - Each core executes a number of *warps* (comparable to a CPU thread)
  - Each warp consists of 32 "threads" that run *in lockstep** (comparable to a single lane of a SIMD execution unit)

*not so for Volta architecture and onwards… http://www.catb.org/jargon/html/W/wheel-of-reincarnation.html

GPU architecture

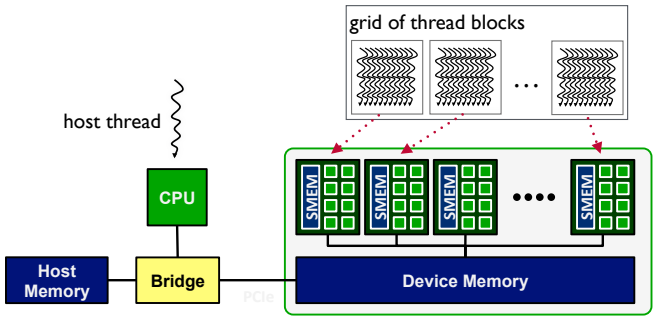
- Each streaming multiprocessor (SM) executes a number of warps
  - The SM has a number of active threads (e.g. Ampere has up to 2048 per SM)
  - The core will switch warps whenever there is a stall in execution (e.g. waiting for memory)
  - Latency is thus hidden by having many active threads; this is only possible if you can feed the GPU enough work

GPU architecture

- There are many similarities between the CPU and GPU
  - Multiple cores
  - A memory hierarchy
  - SIMD vector instructions
- But there are also fundamental differences
  - Each SM executes up to 64 warps, instead of two threads (with SMT2)
  - The memory hierarchy is explicit on the GPU (software managed cache)
  - CPU uses thread (SMTx) and instruction level parallelism to saturate ALUs
  - GPU SIMD is implicit (SIMT model)

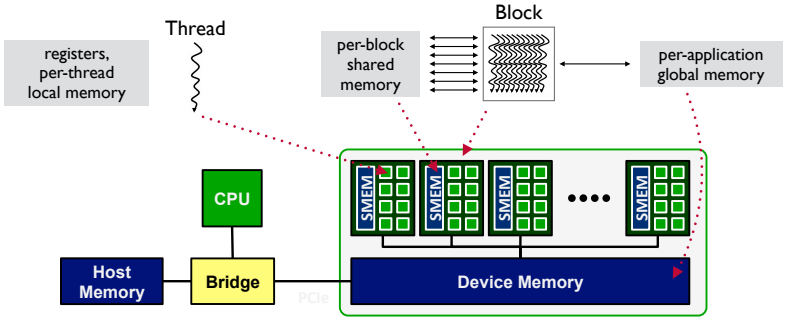
Execution model

- The GPU is a co-processor controlled by a host program
  - The host (CPU) and device (GPU) have separate memory spaces
  - The host program controls data management on the device (allocation, transfer) as well as launching kernels



Execution model

- The GPU kernels execute multiple thread blocks over the SMs
  - All threads execute the same *sequential* program
  - Thread instructions are executed in logical SIMD groups (warps)



Programming model

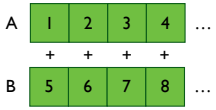
- The CUDA (and OpenCL, Vulkan and Metal) programming model provides
  - A thread abstraction to deal with SIMD
  - Synchronisation and data sharing between *small* groups of threads (100s)
  - A scalable programming model to deal with *lots* of threads (10,000s)
  - A C-like language for device code
    - The similarity is only superficial; it is heavily influenced by the underlying hardware model because people feel more comfortable if there are braces and semicolons ._.

Programming model

- A GPU program consists of the *kernel* run on the GPU
  - Kernels are functions which are executed n times in parallel by n different threads on the device
  - Each thread executes the same *sequential* program
    - We can not execute different code in parallel
- … together with a program on the CPU to launch the kernel and control GPU device operations

Kernels

- Example: element-wise add two vectors



  - Sequential version:

```
void vector_add( float* A, float* B, float* C, int n )
{
   for ( int i = 0; i < n; ++i ) {
      C[i] = A[i] + B[i];
   }
}
```

  - CUDA kernel:

```
__global__ void vector_add( float* A, float* B, float* C, int n )
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if ( i < n ) {
      C[i] = A[i] + B[i];
   }
}
```


Threads

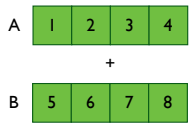
- A kernel consists of multiple copies of the code executed in parallel
  - Each thread has its own registers
  - Each warp or each thread has its own program counter*
  - The order in which threads are executed is not specified
- Threads are very fine-grained
  - Launching threads on the GPU is cheap compared to on the CPU

* Pre-Volta there is one PC per warp; post-Volta each thread has its own PC

Threads

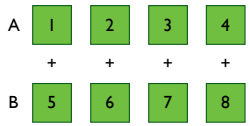
- Threads execute in a single-instruction multiple-thread model (SIMT)
  - In a SIMD model the vector width is explicit
  - In SIMT this is left unspecified
  - Greatly simplifies the programming model

SIMD




```
__m128 a = _mm_set_ps(4, 3, 2, 1);
__m128 b = _mm_set_ps(8, 7, 6, 5);
__m128 c = _mm_add_ps(a, b)
```

SIMT




```
__global__ void vector_add( ... ) {
    // as before
}
```


Threads

- Threads execute in a single-instruction multiple-thread (SIMT) model
  - Understanding how this is mapped to the underlying hardware is important
  - In CUDA threads execute in groups of 32 called a *warp*
  - This is the *logical* vector width
- Performance considerations
  - Threads in a warp share the same program counter
  - Good code will try to keep all threads *convergent* within a warp

Threads

- The scalar (kernel) code is mapped onto the hardware SIMD execution
  - Hardware handles control flow divergence and convergence
  - Divergent control flow between warp threads is handled via an active mask

```
if ( threadIdx.x < 8 ) {
  for ( int i = 0; i < threadIdx.x; ++i ) {
    // ...
  }
}
else
{
  if ( answer == 42 ) {
    // ...
  }
  else {
    // ...
  }
}
```

Threads

- Divergent control flow is handled by predicated execution
  - At each cycle all threads in a warp must execute the same instruction
  - Conditional code is handled by temporarily disabling threads for which the condition is not true (alternatively; false)
  - If-then-else blocks are sequentially executing the 'if' and 'else' branches
- The GPU is therefore a very wide vector processor

Threads

- Divergent control flow is handled by predicated execution
  - Can lead to subtle deadlocks…
  - Consider the canonical implementation of a spin-lock (for the CPU):

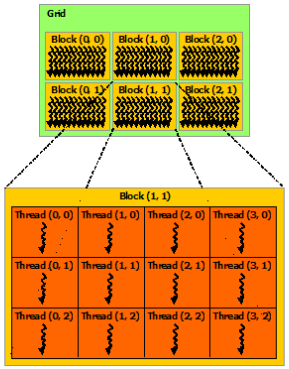
```
do {
    old = atomic_exchange(&lock[i], 1);
} while (old == 1);

/* critical section */

atomic_exchange(&lock[i], 0);
```


Threads

- Benefits of SIMT vs. SIMD
  - Similar to regular scalar code, easier to read and write
- Drawbacks of SIMT vs. SIMD
  - The (logical) vector width is always 32, regardless of the data size
  - Scattered memory access and control flow are not discouraged

Thread hierarchy

- Parallel kernels are composed of many *threads*
  - Executing the same sequential program
  - Each thread has a unique identifier
- Threads are grouped into *blocks*
  - Threads in the same block can cooperate
- A *grid* of thread blocks is the collection of threads which will execute a given kernel
  - Thread blocks will be scheduled onto the SMs of the GPU for execution

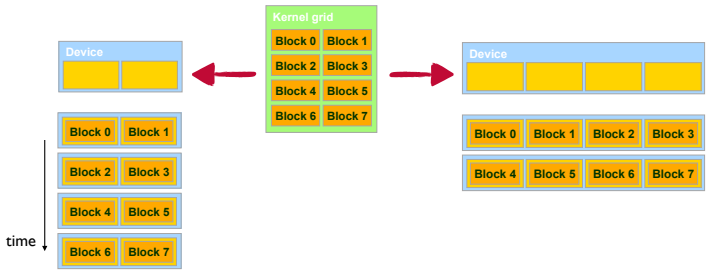


Thread hierarchy

- Individual threads are grouped into thread blocks
  - Each thread block constitutes an *independent data-parallel task*
  - Threads in the same block can cooperate and synchronise with each other
  - Threads in different thread blocks can not cooperate
  - The program must be valid for *any* interleaving of thread blocks
- This independence requirement ensures *scalability*

Thread hierarchy

- Each thread block is mapped onto a SM of the GPU to be executed
  - The hardware is free to assign blocks to any processor (SM) at any time
  - A kernel scales across any number of parallel processors
  - Each block executes in any order relative to other blocks



Thread hierarchy

- Each GPU thread is individually very weak
  - Hardware multithreading is required to hide latency
  - This means that performance depends on the number of thread blocks which can be allocated onto each SM
  - This is limited by the set of registers and shared memory on the SM which are shared between all threads executing on that processor
- Therefore, per-thread resource usage costs performance
  - More registers ⇒ fewer thread blocks
  - More shared (local) memory usage ⇒ fewer thread blocks

Occupancy

- The multiprocessor *occupancy* is the number of kernel threads which can run simultaneously on each SM, compared to the maximum possible
  - Example: Constants for Turing architecture (RTX 2080 and similar)
    - Simultaneous thread blocks (B) ≤ 16
    - Warps per thread block (T) ≤ 32
    - Maximum resident warps: B × T ≤ 32
    - 32-bit registers per thread: B × T × 32 ≤ 65536
    - Shared memory per block (bytes) × B ≤ 65536*
    - Occupancy: B × T / 48

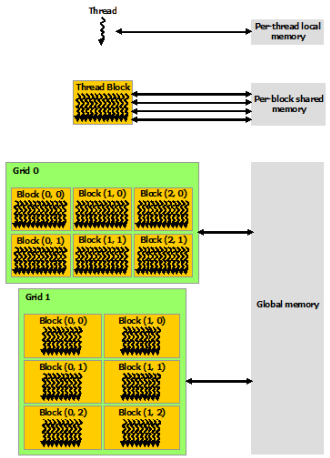
Thread blocks

- Threads in a thread block can communicate and synchronise
  - Example: reverse a vector
  - Question: Does this work?

```
__global__ void reverse( float* arr, int n )
{
    __shared__ float tmp[blockDim.x];
    int gid = blockDim.x * blockIdx.x + threadIdx.x;

    if ( gid < n )
    {
        tmp[threadIdx.x] = arr[gid];
        __syncthreads();
                        = tmp[blockDim.x - threadIdx.x - 1];
    }
}
```

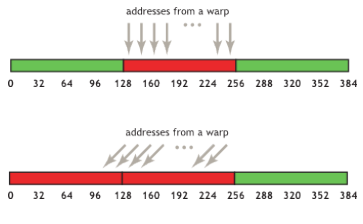

Memory hierarchy

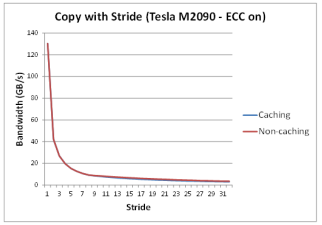
- A many-core processor is a device for turning a compute bound problem into a memory bound problem
  - Lots of processors (ALUs)
  - Memory concerns dominate performance tuning
  - Only global memory is persistent across kernel launches



Memory hierarchy

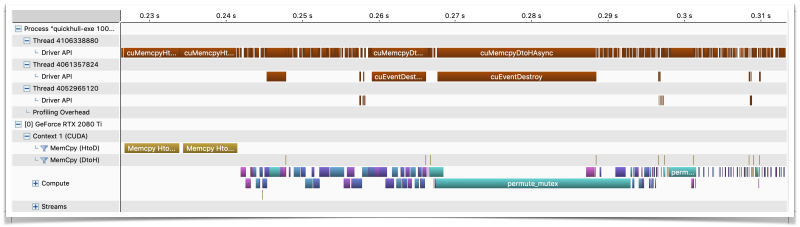
- Global memory is accessed in 32-, 64-, or 128-byte transactions
  - Similar to how a CPU reads a cache line at a time
  - The GPU has a "coalescer" which examines the memory requests from threads in the warp, and issues one or more global memory transactions
- To use bandwidth effectively, threads should read/write in dense blocks





GPGPU

- A typical GPU program
  1. Set up input data on the CPU
  2. Transfer input data to the GPU
  3. Operate on the data
  4. Transfer results back to the CPU
  5. …
  6. profit



Summary

- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access

Summary

- GPUs excel when…
  - The calculation is data-parallel and the control-flow is regular
  - The calculation is large (compute/memory bound)
- CPUs excel when…
  - The calculation is largely serial and the control-flow is irregular
  - The programmer is lazy

Extra slides

- NVIDIA programming guides
- Intel intrinsics guide