B3CC: Concurrency

09: GPGPU

Ivo Gabe de Wolff

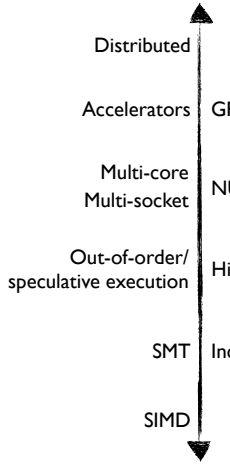
Announcement

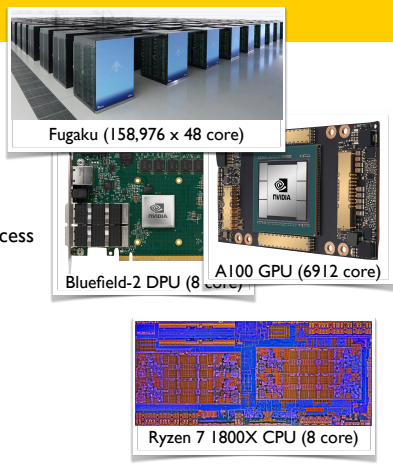
- Mid-term exam next week
  - Tuesday 19-12-2023 @ 13:00 – 15:00 in Olympos Hal 2
  - Covers all the material up to and including STM
  - Excluding Delta-stepping

Announcement

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

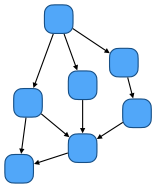
Recap





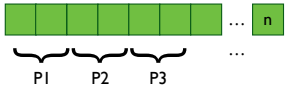
https://en.wikichip.org/wiki/amd/ryzen_7/1800x

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

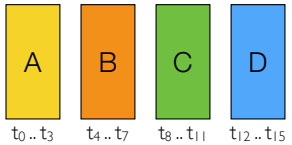
- GPUs are designed to accelerate graphics processing (rasterisation)
  - This is an inherently *data-parallel* task
  - GPUs are designed to maximise *bandwidth*: the time to process as single pixel is less important than the number of pixels processed per second
  - Specialised for compute intensive,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

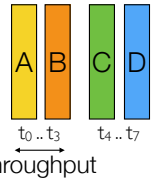
- Image we need to perform some operation that takes 4 units of time (clock cycles), on values A, B, C and D.



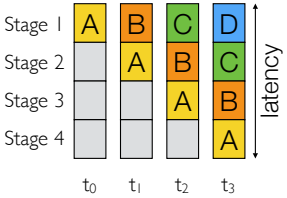
https://en.wikipedia.org/wiki/IP_over_Avian_Carriers

CPU vs. GPU

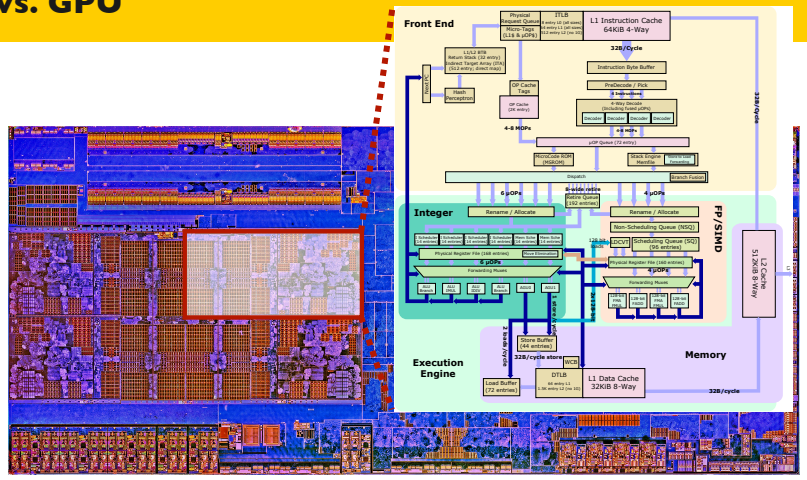
- Horizontal parallelism: increase throughput
  - More execution units working in parallel



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

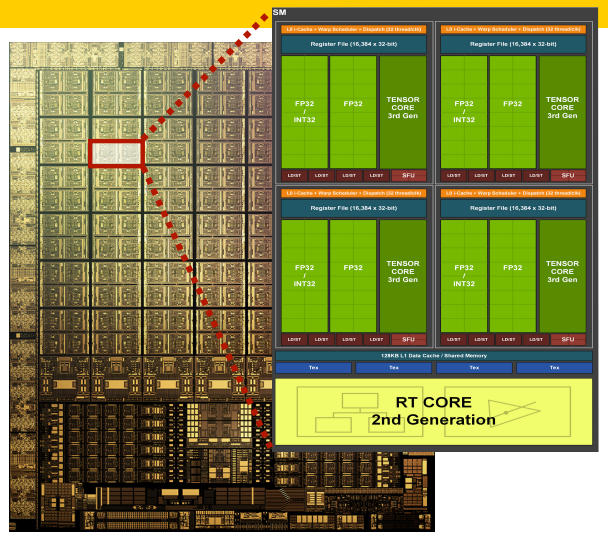


CPU vs. GPU



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

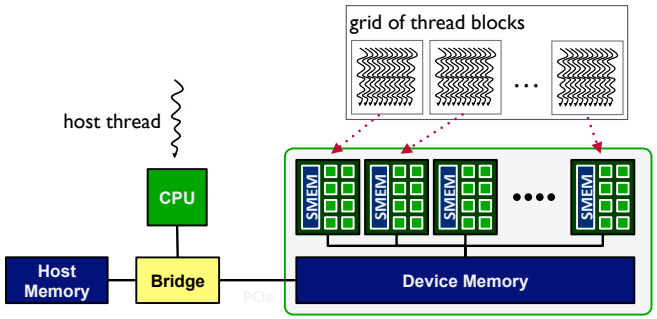
- Each streaming multiprocessor (SM) executes a number of warps
  - The SM has a number of active threads (e.g. Ampere has up to 2048 per SM)
  - The core will switch warps whenever there is a stall in execution (e.g. waiting for memory)
  - Latency is thus hidden by having many active threads; this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

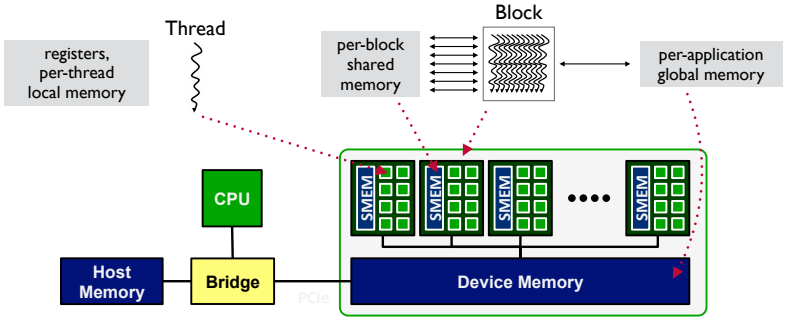
Execution model

- 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

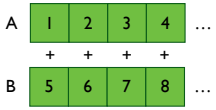
- The CUDA (and OpenCL, Vulkan and Metal) programming model provides
  - A thread abstraction to deal with SIMD
  - Synchronisation and data sharing between *small* groups of threads (100s)
  - A scalable programming model to deal with *lots* of threads (10,000s)
  - A C-like language for device code
    - 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

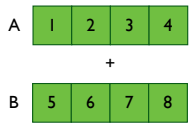
- A kernel consists of multiple copies of the code executed in parallel
  - Each thread has its own registers
  - Each warp or each thread has its own program counter*
  - The order in which threads are executed is not specified
- Threads are very fine-grained
  - Launching threads on the GPU is cheap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

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

Threads

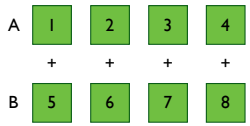
- Threads execute in a single-instruction multiple-thread model (SIMT)
  - In a SIMD model the vector width is explicit
  - In SIMT this is left unspecified
  - Greatly simplifies the programming model

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):

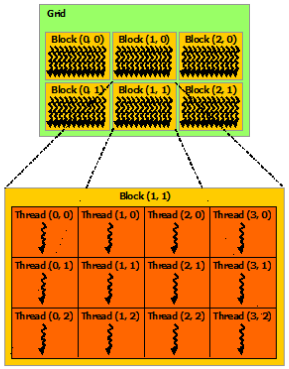
```
do {
    old = atomic_exchange(&lock[i], 1);
} while (old == 1);

/* 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

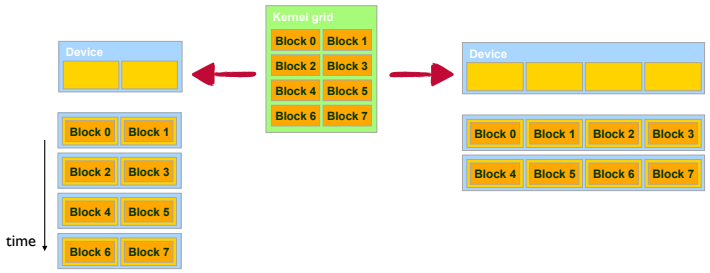


Thread hierarchy

- 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

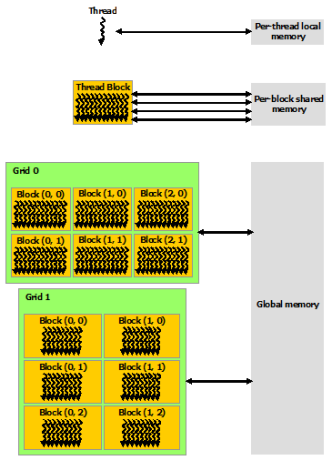
Thread blocks

- 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];
    }
}
```

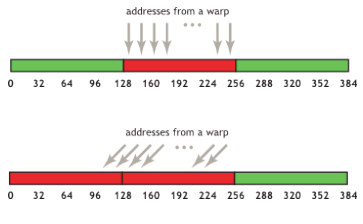

Memory hierarchy

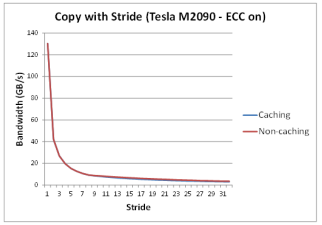
- A many-core processor is a device for turning a compute bound problem into a memory bound problem
  - Lots of processors (ALUs)
  - Memory concerns dominate performance tuning
  - Only global memory is persistent across kernel launches



Memory hierarchy

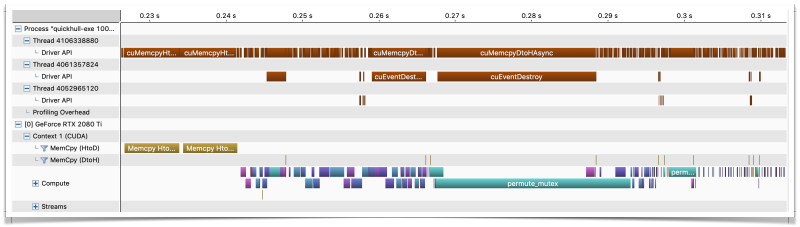
- Global memory is accessed in 32-, 64-, or 128-byte transactions
  - Similar to how a CPU reads a cache line at a time
  - The GPU has a "coalescer" which examines the memory requests from threads in the warp, and issues one or more global memory transactions
- To use bandwidth effectively, threads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

Utrecht University

B3CC: Concurrency

11: Accelerate

Tom Smeding

Scaling and Speedup

Leftovers from 09: Parallelism

Announcement

- Welcome back!
- The third practical is now available
  - Due Friday 26 January @ 23:59
  - You may work in pairs

Speedup

- The performance improvement, or *speedup* of a parallel application, is:
  - Where T_P is the time to execute using P threads/processors

$$\text{speedup} = S_P = \frac{T_1}{T_P}$$

- The *efficiency* of the program is:

$$\text{efficiency} = \frac{S_P}{P} = \frac{T_1}{P\,T_P}$$

- Here, T_1 can be:
  - The parallel algorithm executed on one thread: *relative speedup*
  - An equivalent serial algorithm: *absolute speedup*

Maximum speedup

- Several factors appear as overhead in parallel computations and limit the speedup of the program
  - Periods when not all processors are performing useful work
  - Extra computations in the parallel version not appearing in the sequential version (example: recompute constants locally)
  - Communication time between processes

Amdahl

- The execution time (T_1) of a program splits into:
  - W_{ser}: time spent doing (non-parallelisable) serial work
  - W_{par}: time spent doing parallel work

$$T_P \geq W_{\text{ser}} + \frac{W_{\text{par}}}{P}$$

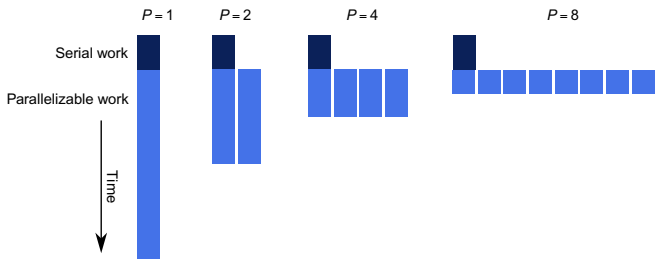
- If $f = \frac{W_{\text{ser}}}{W_{\text{ser}} + W_{\text{par}}}$ is the fraction of serial work to be performed, we get the parallel speedup:

$$S_P \leq \frac{1}{f + (1-f)/P}$$

- This is called *Amdahl's Law*

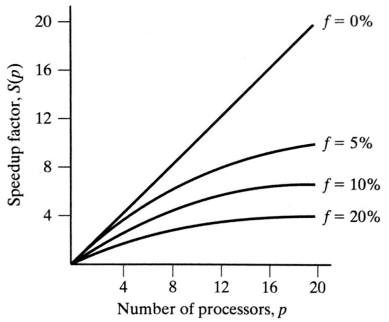
Amdahl

- The speedup bound is determined by the degree of sequential execution in the program, not the number of processors
  - *Strong scaling* (fixed-sized speedup): $\lim_{P\to\infty} S_P \leq 1/f$



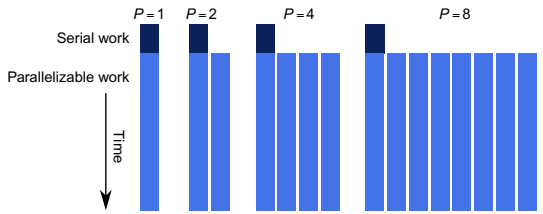
Amdahl

- The serial fraction of the program limits the achievable speedup



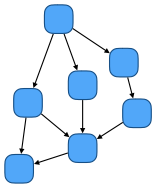
Gustafson-Barsis

- Often the problem size can increase as the number of processes increases
  - The proportion of the serial part decreases
  - *Weak scaling* (scaled speedup): $S'_P = f + (1 - f)P$



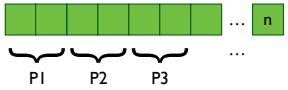
Data parallelism, GPU programming

Recap



Task parallelism

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



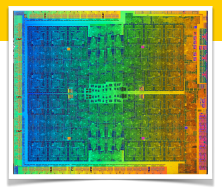
Data parallelism

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

Recap

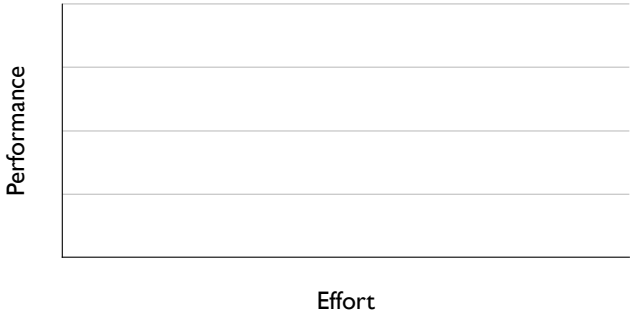
- Despite the name, data parallelism is only a programming model
  - The key is a *single logical thread of control*
  - It does not actually require the operations to be executed in parallel!
  - Today we'll look at a language for data-parallel programming on the GPU

GPU (graphics processing unit)

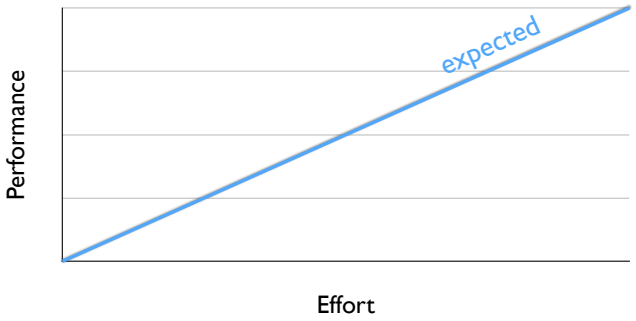


- Lots of interest to use them for non-graphics tasks
  - Machine learning, bioinformatics, data science, weather & climate, medical imaging, computational chemistry, …
  - Can have much higher performance than a traditional CPU
- Specialised hardware with a specialised programming model
  - Caches are software programmable; must be wary of associativity
  - Memory management is explicit, with distinct memory spaces
  - Thousands of threads running simultaneously, each of which can modify any piece of memory at any time

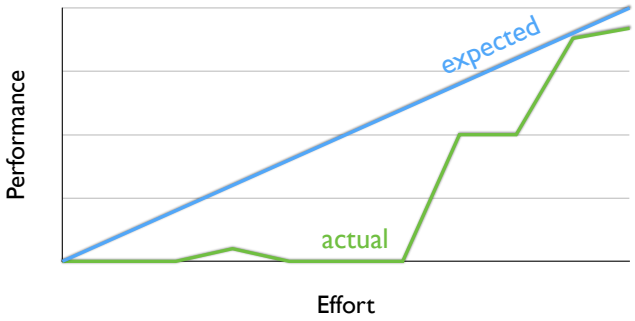
GPU programming



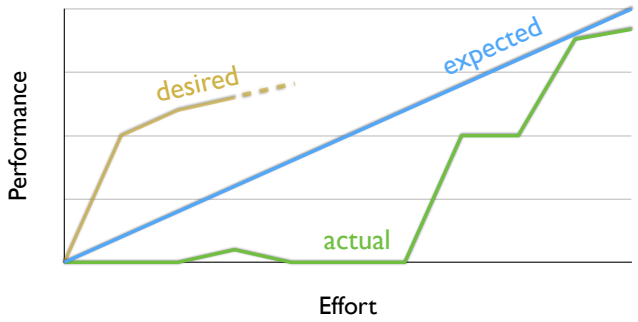
GPU programming



GPU programming

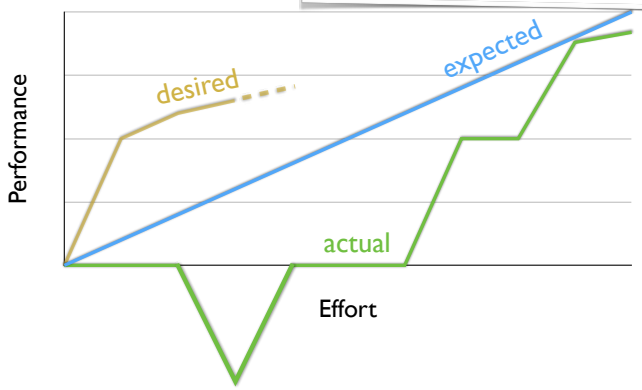


GPU programming



GPU programming

> After expressing available parallelism, I often find that the code has slowed down.
> — Jeff Larkin, NVIDIA Developer Technology



https://devblogs.nvidia.com/getting-started-openacc/

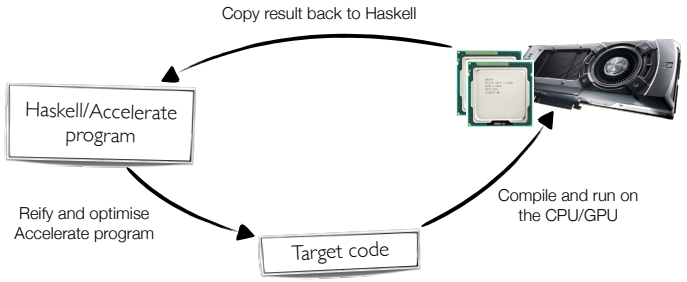
GPU programming

- Two main difficulties:
  1. Structuring the program in a way suitable for GPU parallelisation ←
  2. Writing (performant) GPU code

Accelerate

Accelerate

- An embedded language for *data-parallel arrays* in Haskell
  - Takes care of generating the high-performance CPU/GPU code for us
  - Computations take place on dense multi-dimensional arrays
  - Parallelism is introduced in the form of collective operations on arrays



Accelerate

- Computations take place on arrays
  - Parallelism is introduced in the form of collective operations over arrays
  - map, zipWith, fold, scan (various kinds); permutations (data movement); etc.
  - It is a *restricted* language: consists only of operations which can be executed efficiently in parallel
  - Different types to distinguish parallel computations from scalar expressions

Example: dot product

- In Haskell (lists):

```
import Prelude

dotp :: Num a
     => [a]
     -> [a]
     -> a
dotp xs ys = foldl' (+) 0 (zipWith (*) xs ys)
```

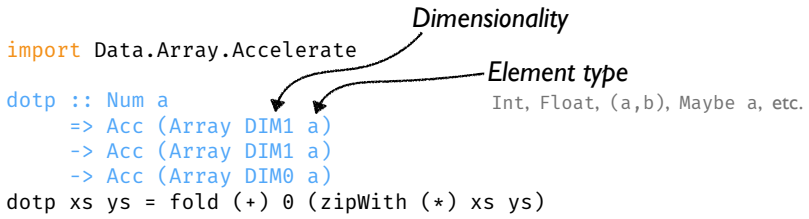

Example: dot product

- In Accelerate:

```
import Data.Array.Accelerate

dotp :: Num a
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Example: dot product

- In Accelerate:




```
import Data.Array.Accelerate

dotp :: Num a
     => Acc (Array DIM1 a)
     -> Acc (Array DIM1 a)
     -> Acc (Array DIM0 a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
Scalar a = Array DIM0 a = Array Z a
Vector a = Array DIM1 a = Array (Z :. Int) a
Matrix a = Array DIM2 a = Array (Z :. Int :. Int) a
           Array DIM3 a = Array (Z :. Int :. Int :. Int) a
             ...
```


Accelerate

- Compile and execute an Accelerate program
  - The same program can be run on different targets

```
import Data.Array.Accelerate.Interpreter
-- import Data.Array.Accelerate.LLVM.Native
-- import Data.Array.Accelerate.LLVM.PTX

run  :: Arrays a => Acc a -> a
runN :: Afunction f => f -> AfunctionR f

runN :: (…) => Acc a -> a
runN :: (…) => (Acc a -> Acc b) -> a -> b
runN :: (…) => (Acc a -> Acc b -> Acc c) -> a -> b -> c
-- ...
```

There's also `runQ`, but don't worry about that

Accelerate

- Parallel computations take place on arrays
  - A *stratified* language of parallel (`Acc`) and scalar (`Exp`) computations
  - Parallel operations consist of many scalar expressions executed in parallel

Accelerate

- The `map` operation:
  - A collective operation (`Acc`) which applies the given scalar function (`Exp`) to each element of the array *in parallel*
  - `map (\x -> x+1) xs` on a one-dimensional array of floats:

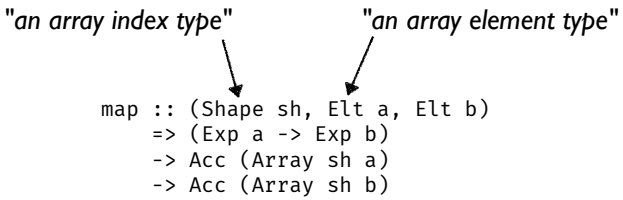
Acc

```
__global__ void map( float* d_xs, float* d_ys, int len )
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if ( i < len ) {
      float x = d_xs[i];
      d_ys[i] = x + 1;
   }
}
```

Exp

Accelerate

- The `map` operation:
  - A collective operation (`Acc`) which applies the given scalar function (`Exp`) to each element of the array *in parallel*



Oddities

- Accelerate is a language *embedded* in Haskell
  - We reuse much of the syntax, but the semantics are different
    - Strict evaluation, unboxed data, no general recursion…
  - Actually, `Acc` and `Exp` are just data structures!
    - Have a `Show` instance
    - The Haskell program *generates* the Accelerate program
    - The `run` operation performs runtime (cross) compilation
  - But the integration has some oddities as well…

Lifting & Unlifting

- Consider the following two types:

```
x :: (Exp Int, Exp Int)
y :: Exp (Int, Int)
```

  - The first is a Haskell pair of embedded expressions on `Int`
  - The second is an embedded expression returning a pair of `Int`s
- How to convert between the two?
  - The pattern synonym `T2`
  - (legacy: the functions `lift` and `unlift` (not recommended))

Pattern synonyms

- We use pattern synonyms for constructing & destructing embedded tuples
  - Can't overload built-in syntax `( , )`, `( , , )`, etc.
  - Instead we use `T2`, `T3`, etc. at both the `Acc` and `Exp` level

```
result :: Acc (Vector Int, Scalar Int)
result = …

T2 idx tot = result
  -- idx :: Acc (Vector Int)
  -- tot :: Acc (Scalar Int)

res = T2 tot idx
  -- res :: Acc (Scalar Int, Vector Int)
```

Shapes

- Array shapes (& indices) are snoc-lists formed from `Z` and `(:.)`
  - `Z` is a zero-dimensional (scalar)
  - `(:.)` adds one inner-most dimension on the right

```
type DIM1     = Z :. Int
type Vector a = Array DIM1 a
```

- More pattern synonyms for constructing & destructing indices

```
x    :: Exp Int
I1 x :: Exp DIM1   -- you'll need this one
```


Pattern matching

- Use the `match` operator to perform pattern matching in embedded code
  - Also note the pattern synonyms for constructing/deconstructing cases

```
foo :: Exp (Maybe Int) -> Exp Int
foo x = x & match \case
  Nothing_ -> 0
  Just_ y  -> y + 1
```


Guards

- Unfortunately guard syntax doesn't work
  - Use a regular if-then-else (chain) instead

```
nope :: Exp Int -> Exp Int
nope x
  | x < 0     = ...
  | otherwise = ...
```


Looping

- Can't write recursive embedded functions directly
  - Need to use an explicit (tail-recursive) looping combinator instead
  - Continue applying the body function (second argument) as long as the predicate function (first argument) returns true

```
awhile
  :: Arrays a
  => (Acc a -> Acc (Scalar Bool))
  -> (Acc a -> Acc a)
  -> Acc a
  -> Acc a
```

Debugging

- Some trace functions for printf-style debugging
  - Output a trace message as well as some arrays to the console before proceeding with the computation
  - Useful for inspecting intermediate values

```
atraceArray
  :: (Arrays a, Arrays b)
  => Text
  -> Acc a
  -> Acc b
  -> Acc b
```

use `"quotes"`

Documentation

- More information in the documentation
  - https://ics.uu.nl/docs/vakken/b3cc/resources/acc-head-docs (latest version, used in the Quickhull template)
  - https://hackage.haskell.org/package/accelerate (released version (older))

Accelerate

- Implementing a data-parallel program consists of two parts:
  - What are the collective (parallel) operations that need to be done?
  - What does each individual (sequential) thread need to do?

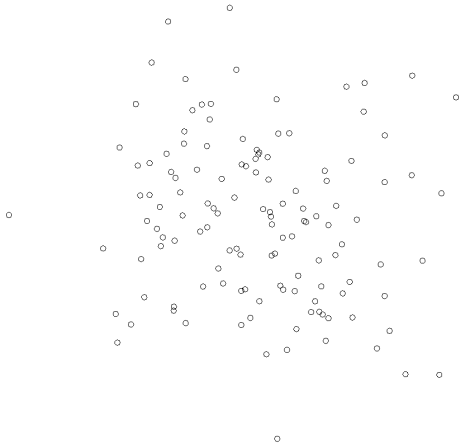
Quickhull

Quickhull

- An algorithm to determine the small polygon containing a set of points
  - You will implement a data-parallel version of the algorithm in Accelerate
  - See the specification for details

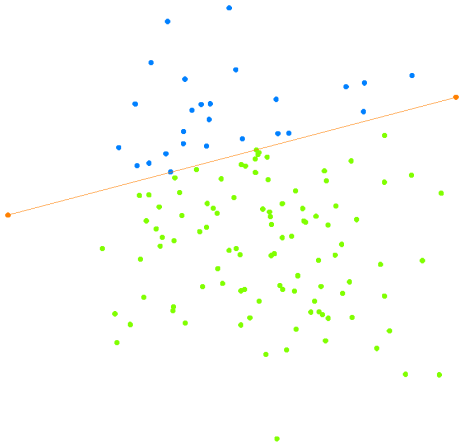
Example

- Initial points
  - The goal is to find the smallest polygon containing all these points
  - This is known as the *convex hull*



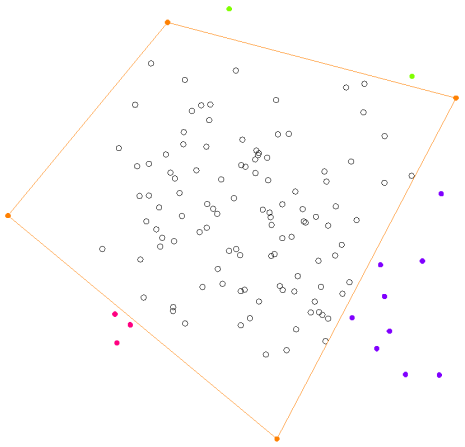
Example

- Create initial partition
  - Choose two points that are definitely on the convex hull
  - Partition others to either side of that line (above/left and below/right)
  - Points of the same colour are in the same segment



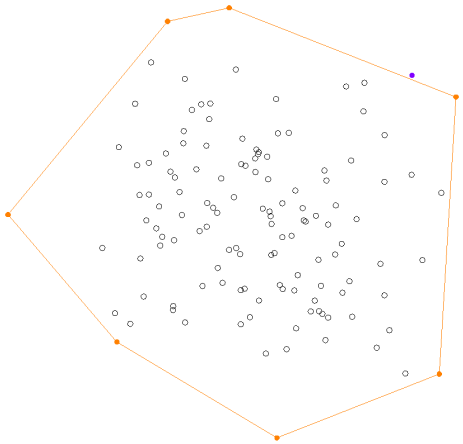
Example

- Recursively partition each segment
  - This is done for all points at once, in data-parallel
  - The hollow circles are points no longer under consideration
  - Orange circles are on the convex hull
  - Other colours are still undecided.
  - Same colours are in the same partition



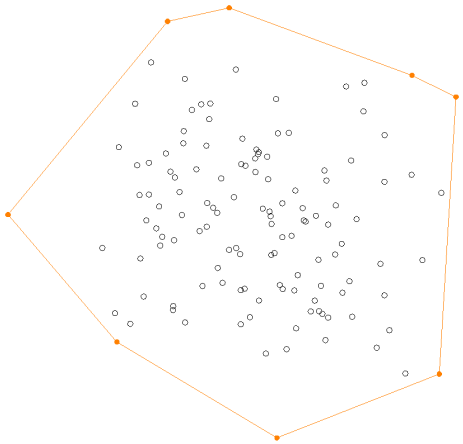
Example

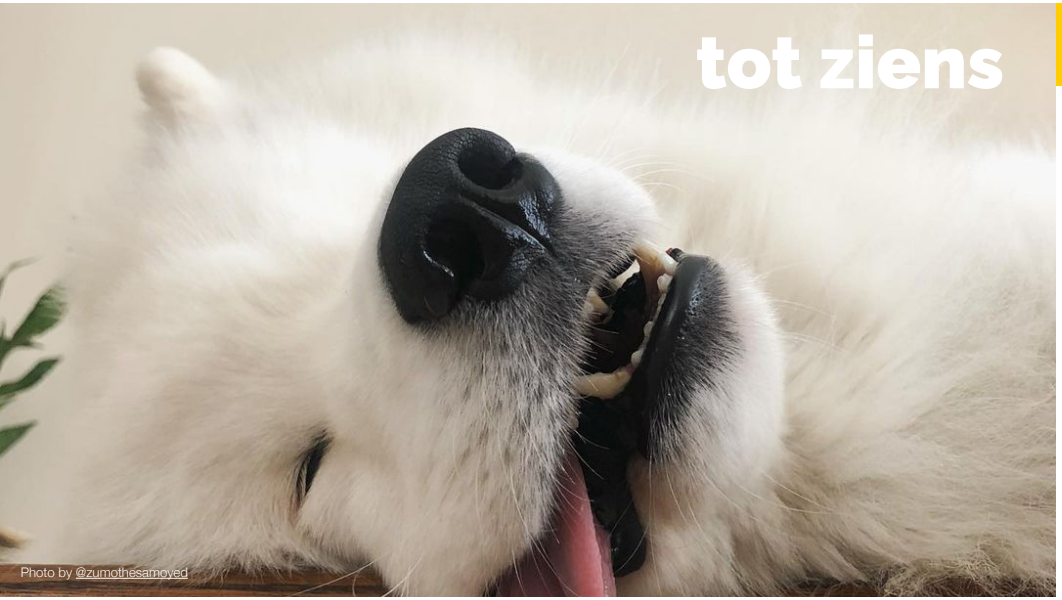
- Continue partitioning each segment…



Example

- … until no undecided points remain





Traditional compiler construction

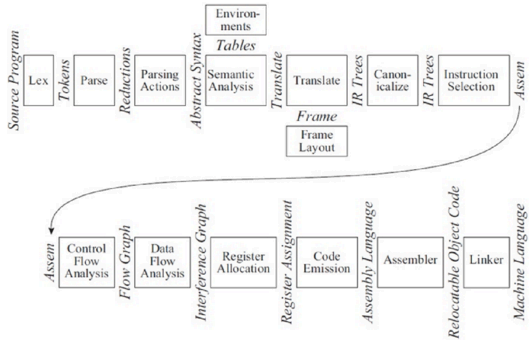


FIGURE 1.1. Phases of a compiler, and interfaces between them

Modern Compiler Implementation in Java, A. Appel and J. Palsberg

Modern compiler construction

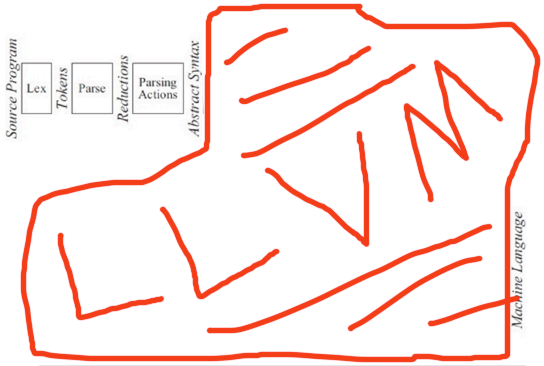


FIGURE 1.1. Phases of a compiler, and interfaces between them

https://msm.runhello.com/p/1003

Utrecht University

B3CC: Concurrency

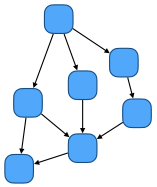
12: Data Parallelism (1)

Ivo Gabe de Wolff

Recap

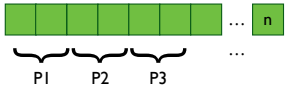
- **Concurrency**: dealing with lots of things at once
- **Parallelism:** doing lots of things at once
  - Processors are no longer getting faster: limitations in power consumption, memory speed, and instruction-level parallelism
  - Adding more processor cores has been the dominant method for improving processor performance for the last decade

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

Goals

- Large applications use a mix of task- and data-parallelism
  - There is a difference in how to make use of 2-4 cores vs. 32+ cores
- In the application of parallelism, we would like to achieve:
  - *Performance*: ease of use, scalability, and predictability
  - *Productivity*: expressivity, correctness, clarity, and maintainability
  - *Portability*: between different machines, compilers, or architectures

Applications

- Games
  - Probably the primary consumer market for teraflop computing applications
- Image and video editing
- Scientific computing
  - Numeric simulations, modelling, etc.
- Machine learning

Patterns

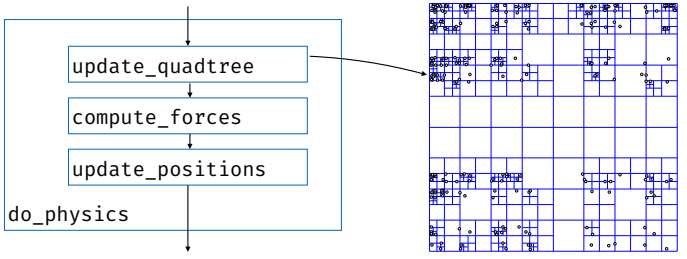
- Patterns, or *algorithmic skeletons*
  - A pattern is a recurring combination of task distribution and data access
  - Patterns provide a vocabulary for [parallel] algorithm design
  - These ideas are not tied to a particular hardware architecture
- This distinguishes two important aspects:
  - Semantics: what we want to achieve
  - Implementation: how to achieve this on a given architecture

Patterns

- Patterns also exist in serial code
  - We often don't think of serial code in this way, however it helps to name these patterns in order to talk about these ideas in a parallel context
  - Compositional patterns: nesting
  - Control-flow patterns: sequence, selection, repetition, and recursion

Patterns: nesting

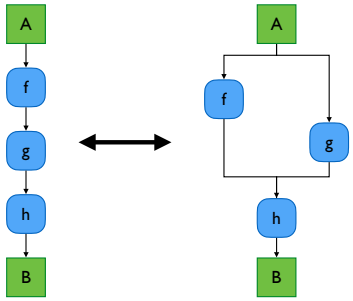
- Nesting simply refers to the ability to hierarchically compose patterns
  - Including recursive functions



Patterns: sequence

- Tasks executed in a specified order
  - We generally assume that the program is executed in the text order
  - Modern CPUs violate this (out-of-order execution (instructions & memory))
  - Programmer or language specifies if/how memory operations may be reordered (memory_order in c++)

```
f = a * a
g = a + a
h = f - g
```

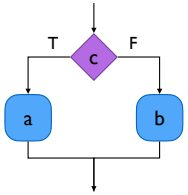


https://en.wikipedia.org/wiki/Out-of-order_execution
https://en.wikipedia.org/wiki/Memory_ordering#Runtime_memory_ordering
https://en.cppreference.com/w/cpp/atomic/memory_order

Patterns: selection

- Conditionals are pervasive in serial code
  - On average one branch every five instructions
  - Modern CPUs speculatively execute (far) ahead of when `c` is known
  - TensorFlow (google deep learning framework) always evaluates both branches of conditionals

```
if (c) {
  a;
} else {
  b;
}
```

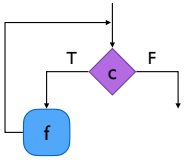


https://en.wikipedia.org/wiki/Speculative_execution

Patterns: iteration

- Continually execute a task while some condition is true
  - Parallelisation of loops is complicated due to *loop-carried dependencies*
  - There is a lot of research in this area (polyhedral models, loop nests)
  - Instead, several *parallel* patterns exist for specific loop forms: map, reduce, scan, scatter, gather…

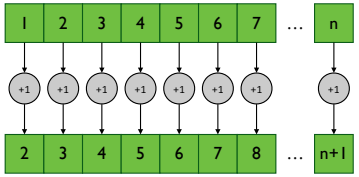
```
while (c) {
  f;
}
```



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

Map

- The `map` operation applies the *same* function to each element of a set
  - This is a parallelisation of a loop with a *fixed* number of iterations
  - There must not be any dependencies between loop iterations: the function uses only the input element value and/or index

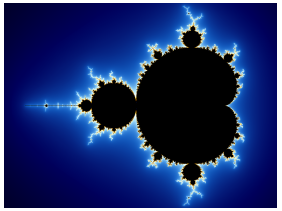



```
for (i = 0; i < len; ++i)
{
   x = xs[i];
   y = f(x);
   ys[i] = y;
}
```

https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:18

Map

- The map operation applies the *same* function to each element of a set
  - The function only has access to a *single value*
  - The number of iterations is dynamic (e.g. size of the array) but fixed at the start of the `map`: does not vary based on the loop body
  - Note that the order of operations is not specified



$$z_{n+1} = z_n^2 + c$$

Map

- The map operation applies the *same* function to each element of a set
  - On the GPU this corresponds to one thread per element
  - Number of loop iterations is controlled by how many threads the kernel is launched with
  - Host code:

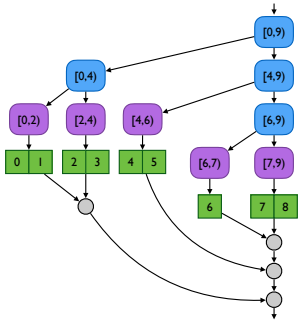
```
map<<<4, 1024>>>( h_xs, h_ys, 4000 );
```

  - GPU code:

```
__global__ void map( float* d_xs, float* d_ys, int len )
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if ( i < len ) {
      d_ys[i] = f ( d_xs[i] );
   }
}
```

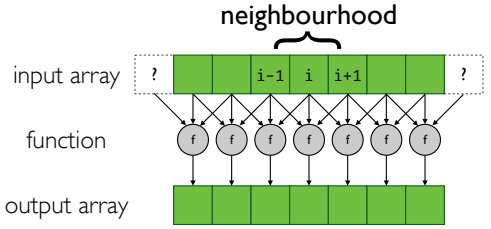

Map

- In the graphics pipeline, vertex and fragment shaders are a parallel map
  - Each shader outputs a single pixel or vertex; no other side effects
  - Shaders are also examples of *streaming algorithms*: data is used exactly once, so no caching is necessary
- On the CPU, can be implemented via
  - Static schedule (like count & list mode of IBAN)
  - fork-join
  - divide-and-conquer (like search mode of IBAN)
  - …



Stencil

- A `map` with access to the neighbourhood around each element
  - The set of neighbours is fixed, and relative to the element
  - Ubiquitous in scientific, engineering, and image processing algorithms



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:37

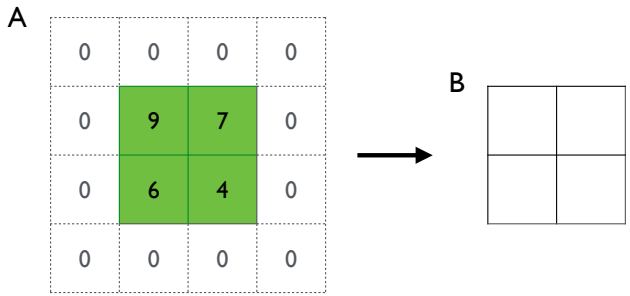
Stencil

- The stencil pattern
  - The set of neighbouring elements used by the stencil function
  - The shape of the stencil pattern can be anything: sparse, non-symmetric, etc.
  - The pattern of the stencil determines how the stencil operates in an application

```
__global__ void stencil( float* xs, float* ys, int width, int height )
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   ys[i] = f (          xs[i-width]
             , xs[i-1], xs[i],      xs[i+1]
             ,          xs[i+width]
             );
}
```

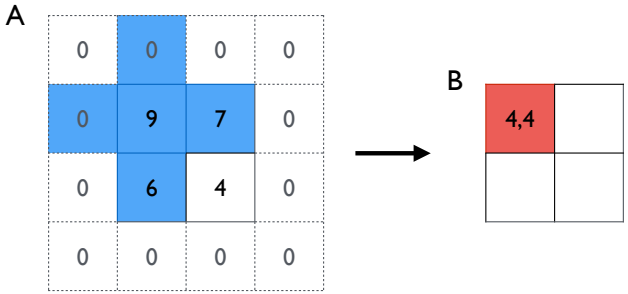

Example

- Apply a stencil operation to the inner square
  - Treat out-of-bounds elements are zero (we'll come back to this later)



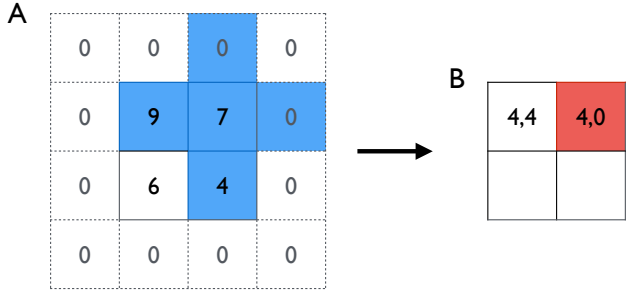
Example

- Apply a stencil operation to the inner square
  - Treat out-of-bounds elements are zero
  - Stencil function: average of the blue squares



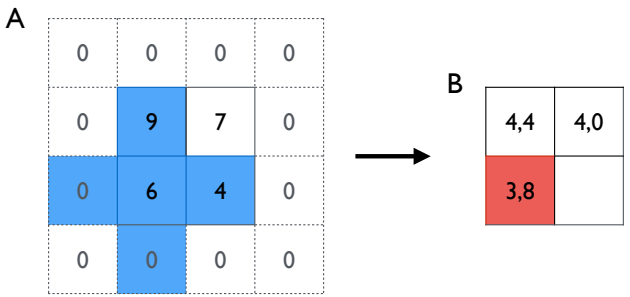
Example

- Apply a stencil operation to the inner square
  - Treat out-of-bounds elements are zero
  - Stencil function: average of the blue squares



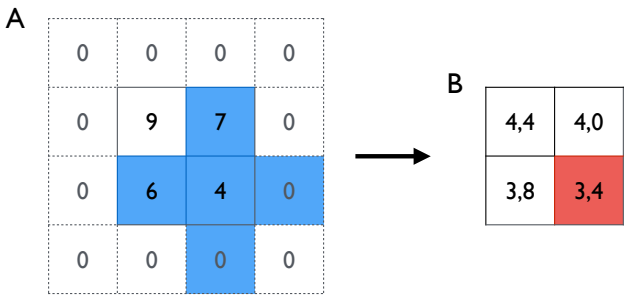
Example

- Apply a stencil operation to the inner square
  - Treat out-of-bounds elements are zero
  - Stencil function: average of the blue squares



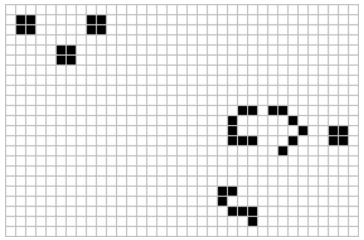
Example

- Apply a stencil operation to the inner square
  - Treat out-of-bounds elements are zero
  - Stencil function: average of the blue squares



Example: Conway's game of life

- Cellula automaton developed in 1970
  - Evolution of the system is determined from an initial state
  - Cells live or die based on the population of their surrounding neighbours
  - Turing complete!

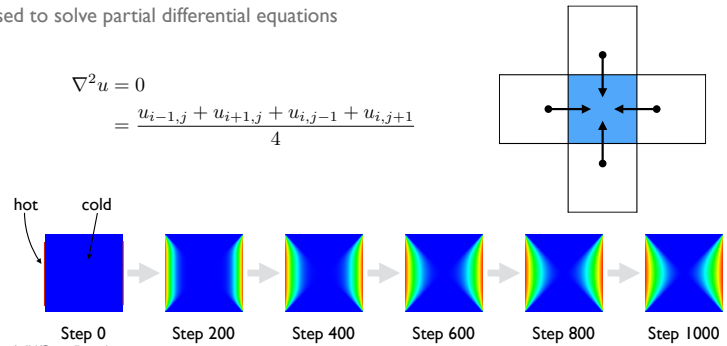


https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
https://github.com/tmcdonell/gameoflife-accelerate

Example: heat equation

- Iterative codes are ones that update their data in steps
- Most commonly found in simulations for scientific & engineering applications
  - Often used to solve partial differential equations

$$\nabla^2 u = 0$$
$$= \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4}$$

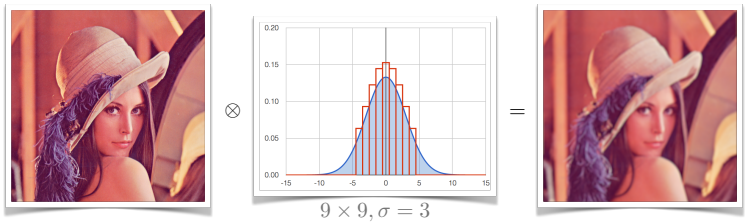


https://en.wikipedia.org/wiki/Stencil_code
https://github.com/tmcdonell/fisher-accelerate

Example: gaussian blur

- Convolution with a Gaussian function
  - Typically used to reduce image noise
  - Each pixel becomes the weighted sum of the surrounding pixels

$$(I \otimes K)(x, y) = \sum_i \sum_j I(x + i, y + j) K(i, j)$$



$9 \times 9, \sigma = 3$

https://en.wikipedia.org/wiki/Gaussian_blur
https://github.com/tmcdonell/accelerate-examples/blob/master/examples/canny/src-acc/Canny.hs#L82

Example: gaussian blur

- Gaussian function
  - This is a *separable* convolution:
    instead of a single $n \times n$ stencil,
    it can be implemented as an $1 \times n$ stencil after a $n \times 1$ stencil
  - This is significant for large n
  - Example: 3×3 stencil

$$\begin{bmatrix} 0.077847 & 0.123317 & 0.077847 \\ 0.123317 & 0.195346 & 0.123317 \\ 0.077847 & 0.123317 & 0.077847 \end{bmatrix} = \begin{bmatrix} 0.27901 \\ 0.44198 \\ 0.27901 \end{bmatrix} \times \begin{bmatrix} 0.27901 & 0.44198 & 0.27901 \end{bmatrix}$$

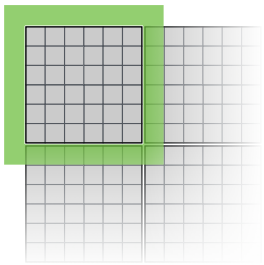
http://dev.theomader.com/gaussian-kernel-calculator

Stencil boundary

- What to do when the stencil pattern falls outside the bounds of the array?
  - At the edges of a simulation, we may need to impose boundary conditions
    - choose a fixed value or derivative (e.g. to impose symmetry)
    - many options are possible…
- What about between processors?

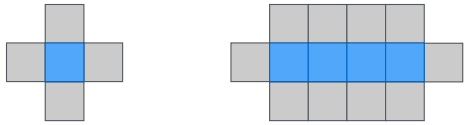
Stencil boundary

- What happens at the boundary of the computation?
  - Each larger box is owned by a thread / processor
- *Ghost cells* are one solution to the boundary and update issues of a stencil computation
  - Each thread keeps a copy of the neighbour's data to use in local computations
  - The ghost cells must be updated after each iteration of the stencil
  - The set of ghost cells is called the *halo*
    - A deeper halo can be used to reduce communication for some redundant work



Stencil optimisations

- Use a different kernel for the interior and border regions
  - In the gaussian blur example of a 512x512 pixel image, 98% of the pixels do not require in-bounds checks
- Optimise data locality & reuse through tiling
  - *Strip mining* is an optimisation that groups elements in a way that avoids redundant memory access and aligns accesses with cache lines



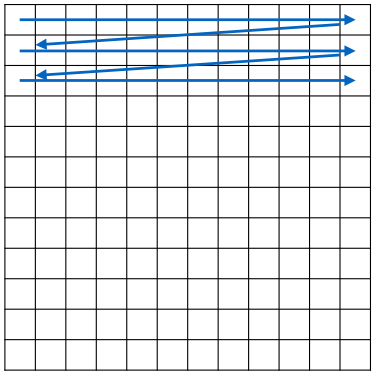
4 x (5 reads + 1 write)

14 reads + 4 writes

Stencil optimisations

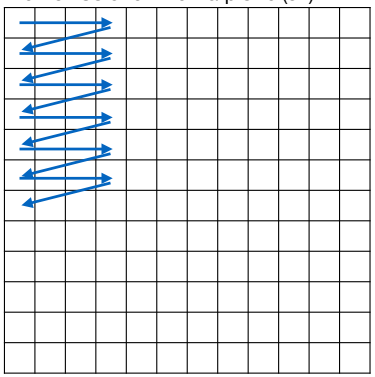
Without tiling

- When handling row 0, row 1 is loaded in cache.
- First values of row 1 may already be out of cache, when handling row 1

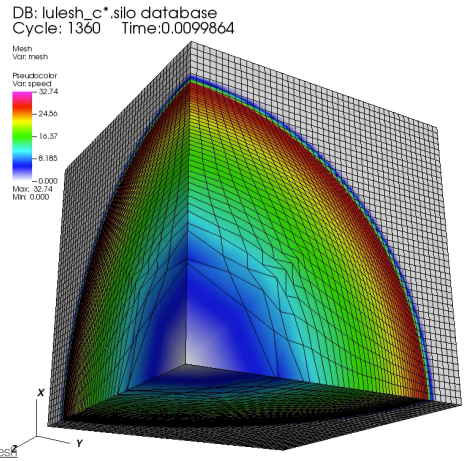


With tiling

- Previously loaded row is still in cache
- Tile width is usually a power of 2, on GPUs often the warp size (32)



Example: LULESH



https://computing.llnl.gov/projects/co-design/lulesh
https://github.com/tmcdonell/lulesh-accelerate

Summary

- Data-parallelism is a good fit for parallel computing
  - Conceptually simple programming model: single logical thread of control
  - Separate the pattern (what you want to do) from the implementation (how to do it: optimisations, target hardware, etc.)


tot ziens

Utrecht University

B3CC: Concurrency

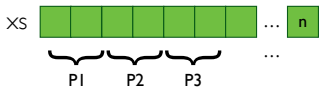
13: Data Parallelism (2)

Ivo Gabe de Wolff

Recap

- Data parallelism: well understood & supported approach to massive parallelism

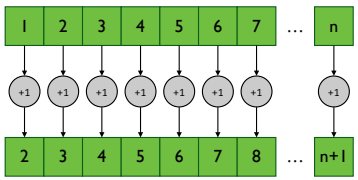
```
parallel_for (i = 1..N) {
    // ... do something to xs[i]
}
```



- Single point of concurrency
- Easy to implement: well supported (Fortran, MPI, OpenMP…), scales to large number of processors, etc.
- Good cost model (work & span): conceptually very simple!
- BUT! the "something" has to be sequential

Recap

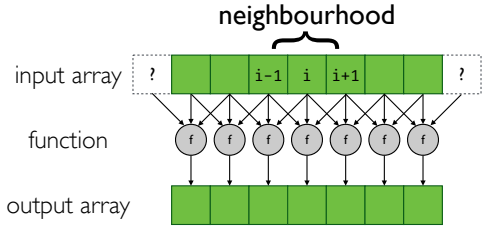
- The `map` operation applies the *same* function to each element of a set
  - This is a parallelisation of a loop with a *fixed* number of iterations
  - There must not be any dependencies between loop iterations: the function uses only the input element value and/or index




```
for (i = 0; i < len; ++i)
{
  x = xs[i];
  y = f(x);
  ys[i] = y;
}
```

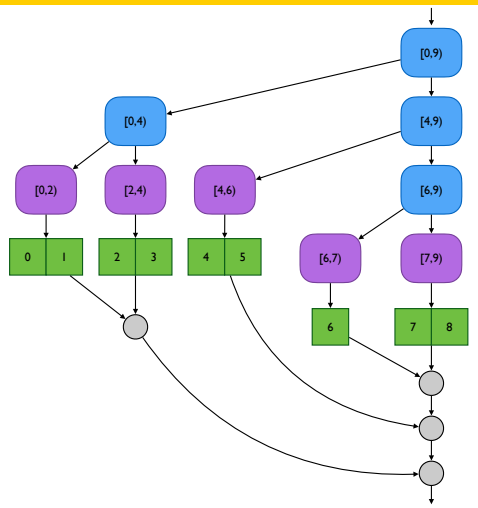

Recap

- A `map` with access to the neighbourhood around each element
  - The set of neighbours is fixed, and relative to the element
  - Ubiquitous in scientific, engineering, and image processing algorithms



Data parallelism on CPUs

- Distribute work via
  - Static schedule (like count & list mode of IBAN)
  - fork-join
  - divide-and-conquer (like search mode of IBAN)
  - …



Data parallelism on GPUs

- A GPU program consists of the kernel that runs on the GPU
  - Kernel functions are executed n times in parallel by n different threads
  - Each thread executes the same sequential program
  - Each thread can distinguish itself from all others *only* by it's thread identifier
    - Any information a thread needs should be directly derivable from this ID

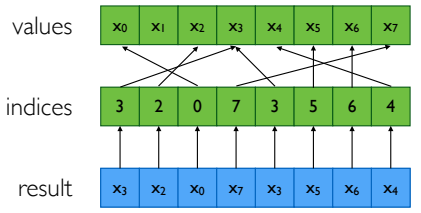
```
__global__ void kernel( float* xs, float* ys, int n, ... )
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      // do something
   }
}
```


More parallel patterns

- We have seen:
  - Map
  - Stencil
- We will discuss today and next time:
  - Gather or backwards permute: random reads
  - Scatter or permutation: random writes
  - Fold or reduction: combined value of all items
  - Scan prefix sum: at each index, combined value of all prior elements

Gather

- The *gather* pattern performs independent random *reads* in parallel
  - Also known as a *backwards permutation*
  - Collects all the data from a source array at the given locations

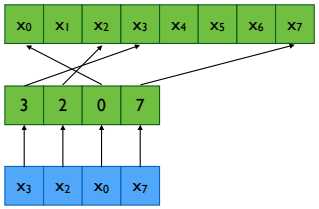


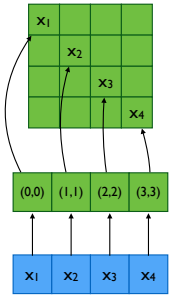

```
for (i = 0; i < len; ++i)
{
   idx = indices[i];
   val = values[idx];
   result[i] = val;
}
```

https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:29

Gather

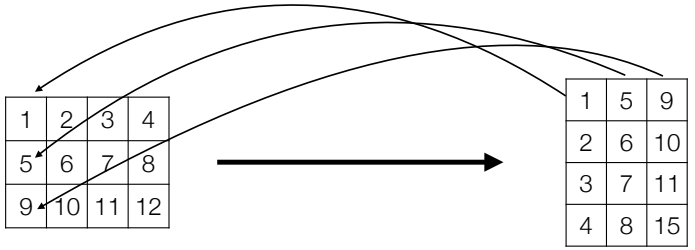
- The *gather* pattern performs independent random *reads* in parallel
  - Requires a function from output index to input index
  - Not all input values have to be read
  - Some values may be read twice
  - Input and output may have different dimensions





Example: matrix transpose

- Transpose rows and columns of a matrix

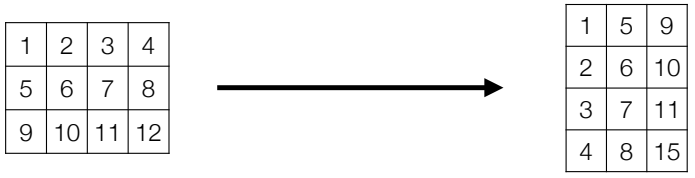


Example: matrix transpose

- Transpose the rows and columns of a matrix

```
transpose :: Elt a ⇒ Acc (Matrix a) → Acc (Matrix a)
transpose xs =
  let I2 rows cols = shape xs
   in backpermute (I2 cols rows) (\(I2 y x) → I2 x y) xs


__global__ void transpose( float* xs, float* ys, int rows, int cols)
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      int row = idx / rows;
      int col = idx % cols;
      ...
   }
}
```

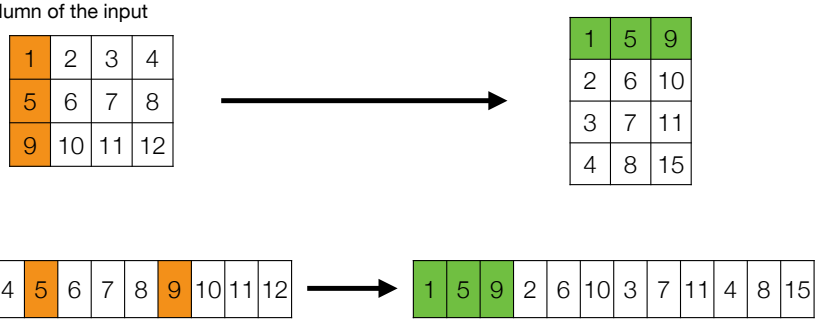

Example: matrix transpose



- In memory, this is stored as:

Example: matrix transpose

- To write one row of the output, we read one column of the input



Example: matrix transpose

- The memory access pattern for transpose is not ideal
  - On the CPU work in tiles to improve cache behaviour
  - On the GPU use shared memory explicitly to do coalesced reads & writes

Example: matrix vector multiply

- The *dense* matrix-vector multiply
  - Perform a dot-product of each row of the matrix against the vector
  - Can be parallelised in different ways

```
for (r = 0; r < rows; ++r) {
  result[r] = 0;
  for (c = 0; c < cols; ++c) {
    // dot product of this row with the vector
    result[r] += matrix[r][c] * vector[c];
  }
}
```


Example: sparse-matrix vector multiply

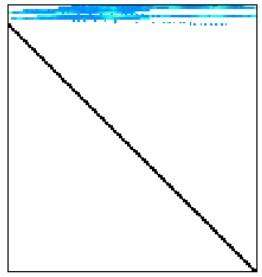
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 11 \\ 0 \\ 1 \\ 9 \end{pmatrix}$$

Example: sparse-matrix vector multiply

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 11 \\ 0 \\ 1 \\ 9 \end{pmatrix}$$

Example: sparse-matrix vector multiply

- Multiply a *sparse* matrix by a dense vector
  - Example: Hardesty3 dataset
    - Matrix size is 8.2M x 7.6M
    - Only 40M non-zero entries (0.000065%)
  - Want to store only the non-zero entries, as only these will contribute to the result
  - Together with the row/column index of each element (various encodings possible)



https://sparse.tamu.edu/Hardesty/Hardesty3

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)
  - Stores only the non-zero elements together with their column index
  - Also need the number of non-zero elements in each row
  - …corresponds to:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

index-value pairs `[ (0, 1.0), (1, 1.0), (1, 7.0), (2, 3.0), (3, 2.0) , (1, 1.0), (3, 3.0), (4, 4.0) ]`

segment descriptor `[ 2, 3, 0, 1, 2 ]`

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)
  - Stores only the non-zero elements together with their column index
  - Also need the number of non-zero elements in each row
  - …corresponds to:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

index-value pairs `[ (0, 1.0), (1, 1.0), (1, 7.0), (2, 3.0), (3, 2.0) , (1, 1.0), (3, 3.0), (4, 4.0) ]`

segment descriptor `[ 2, 3, 0, 1, 2 ]`

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

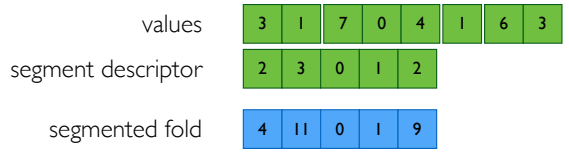
indices	`[ 0, 1, 1, 2, 3, 1, 3, 4 ]`
values	`[ 1.0, 1.0, 7.0, 3.0, 2.0, 1.0, 3.0, 3.0 ]`
segment descriptor	`[ 2, 3, 0, 1, 2 ]`
vector	`[ 3, 1, 0, 2, 1 ]`

- The sparse-matrix dense-vector multiply is then:
  1. *gather* the values from the input vector at the column indices
  2. pair-wise multiply (1) with the matrix values (*zipWith*)
  3. segmented *reduction* of (2) with the matrix segment descriptor
     - *… more on reductions and segmented operations next time!*

https://github.com/tmcdonell/accelerate-examples/tree/master/examples/smvm

Example: sparse-matrix vector multiply

- This can be viewed as a kind of *nested* data-parallel computation: parallel computations which spawn further parallel work
  - More difficult to parallelise (for both hardware and software)
  - Segmented operators allow us to convert nested parallel computations into *flat* parallel computations

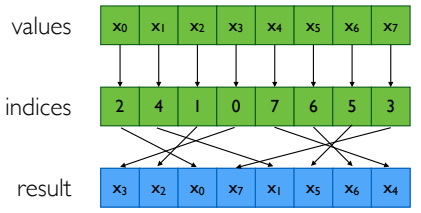


Gather

- Gather or backwards permutation transforms indices in the *output* array to indices in the *input* array
  - But; arbitrary memory access patterns are slow (especially on the GPU)
  - Simple pattern; many common cases which can be made more efficient
- Next is scatter, forward permutation, which transforms indices in the *input* array to indices in the *output* array

Scatter

- The *scatter* pattern performs independent random *writes* in parallel
  - Also known as forward permutation
  - Puts data from the source array into the specified locations




```
for (i = 0; i < len; ++i)
{
   val = values[i];
   idx = indices[i];
   result[idx] = val;
}
```

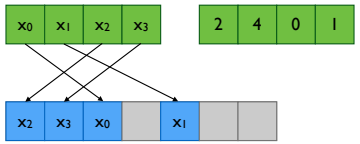
https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:28

Scatter

- The *scatter* pattern performs independent random *writes* in parallel
  - Analogously to gather, we can consider scatter as an index mapping *f* transforming indices in the *input* (source) array to indices in the *output* (destination) array
  - More complex than gather, especially if
    - *f* is not surjective: the range of *f* might not cover the entire codomain
    - *f* is not injective: distinct indices in the domain may map to the same index in the codomain
    - *f* is partial: elements in the domain may be ignored

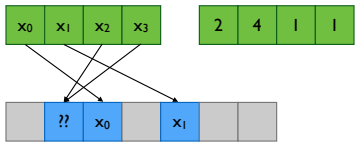
Scatter

- The index permutation might not cover every element in the output
  - We need to first initialise the output array



Collisions

- Multiple values may map to the same output index
  - Possible strategies to handle *collisions:*
    - Disallow
    - Non-deterministically, one write succeeds
    - Merge values with a given associative and commutative operation



Collisions: atomic instructions

Possible strategies to handle *collisions:*

1. Non-deterministically, one write succeeds
   - Requires atomic writes
   - Writes of single words are typically atomic, but that depends on architecture
2. Merge values with a given associative and commutative operation
   - Use an atomic read-modify-write instruction (e.g. atomic_fetch_add), if it exists for this operation
   - Use an atomic compare-and-swap loop, if a value is a single word
     - Maximal size of a word for compare-and-swap depends on the architecture
3. Use (per element) locks otherwise

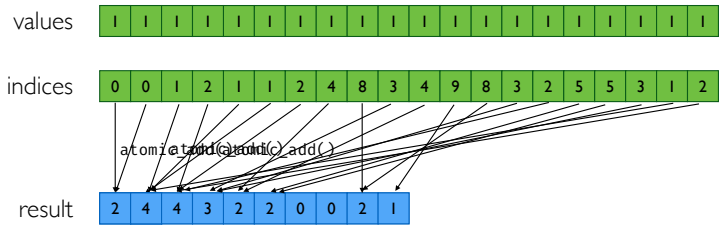
Collisions: locks

- A general merge function might need to implement some locking strategy
  - If no atomic instruction exists; or multiple words are updated
  - Recall: this classic spin lock executed on the GPU can deadlock:

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

/* critical section */

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


Example: histogram

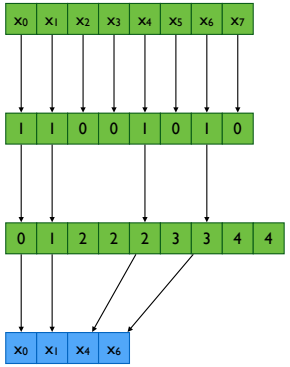
- Computing a histogram requires merging writes to the same location
- Sample data: `[0,0,1,2,1,1,2,4,8,3,4,9,8,3,2,5,5,3,1,2]`



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#v:permute

Example: filter (compact)

- Return only those elements of the array which pass a predicate
  1. *map* the predicate function over the values to determine which to keep
  2. exclusive *scan* the boolean flags to determine the output locations and number of elements to keep
  3. *permute* the values into the position given by (2) if (1) is true



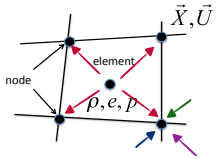
https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#v:filter

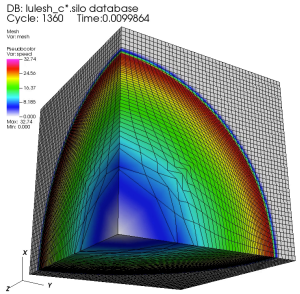
Scatter

- Scatter is more expensive than gather for a number of reasons
  - Not only to handle collisions!
  - Due to the behaviour of caches, there is inter-core communication when threads access the same cache *line*, even if there is no actual collision
  - If the target locations are known in advance, scatter can be converted into a gather operation (this may require extra processing)

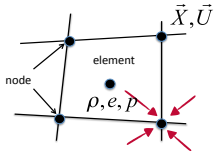
Scatter

- Reframing an algorithm can be key to converting scatter to gather
  - As always, there are different tradeoffs in computation vs. communication
  - Per element: scatter





  - Per node: gather



Summary

- Performance is often more limited by data movement than computation
  - Transferring data across memory layers is costly
  - Data organisation and layout can help to improve locality & minimise access times
  - Design the application around the data movement
- Similar consistency issues arise as when dealing with computation parallelism
- Might involve the creation of additional intermediate data structures
- Some applications are all about data movement: searching, sorting…

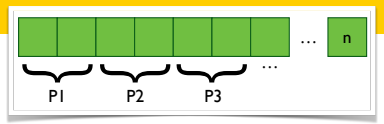
Utrecht University

B3CC: Concurrency

14: Data Parallelism (3)

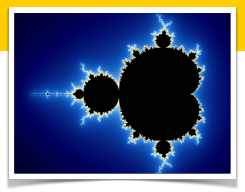
Ivo Gabe de Wolff

Recap



- Data parallelism: well understood approach to massive parallelism
  - Distributes the *data* over the different processing nodes
  - Executes the *same* computation on each of the nodes (threads)
  - Scales to very large numbers of processors
  - Conceptually simple: single thread of control

Recap



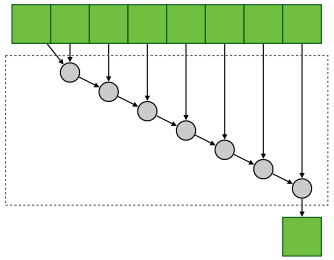
- So far our parallel patterns are *embarrassingly parallel*
  - Each operation is completely independent* from the computation in other threads
- But some collective operations deal with the data as a whole
  - The computation of each output element may depend on the results at other outputs (computed by other threads)
  - More difficult to parallelise!

```
__global__ void kernel( float* xs, float* ys, int n, ... )
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      // do something & communicate with others
   }
}
```


Fold

- Combine a collection of elements into a single value
  - A function combines elements pair-wise
  - Example: sum, minimum, maximum

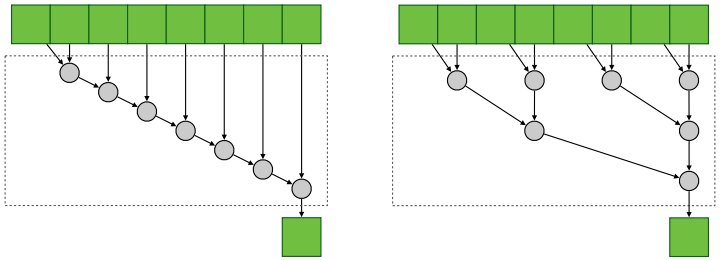
```
// fold1 (n > 0)
r = x[0];
for (i = 1; i < n; ++i)
  r = combine(r, x[i]);

// fold (n ⩾ 0)
r = initial_value;
for (i = 0; i < n; ++i)
  r = combine(r, x[i]);
```



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:32

Fold

- Parallel reduction changes the order of operations
  - Number of operations remains the same, using $\lceil \log_2 N \rceil$ steps



Sequential

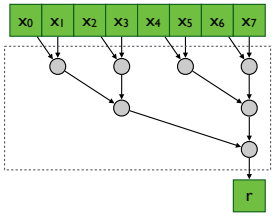
Parallel

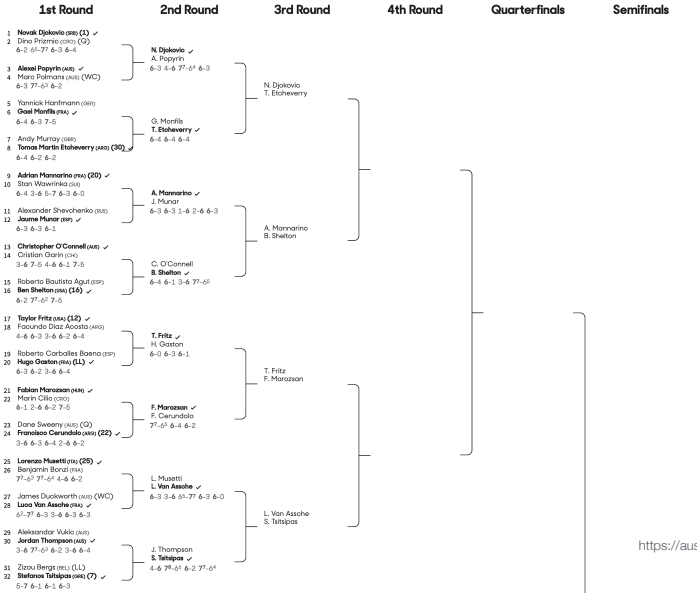
Fold

- Parallel reduction changes the order of operations
  - In order to do this, the combination function must be associative

$$
\begin{aligned}
r &= x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4 \otimes x_5 \otimes x_6 \otimes x_7 \\
&= ((((((x_0 \otimes x_1) \otimes x_2) \otimes x_3) \otimes x_4) \otimes x_5) \otimes x_6) \otimes x_7 \\
&= ((x_0 \otimes x_1) \otimes (x_2 \otimes x_3)) \otimes ((x_4 \otimes x_5) \otimes (x_6 \otimes x_7))
\end{aligned}
$$

  - Other optimisations are possible if the function is commutative, or the initial value is an identity element
  - In general difficult to automatically prove these properties for user defined functions





https://ausopen.com/draws#!mens-singles

Fold in tournaments

- Australian Open has 128 participants
- Fold "computes" the best or maximum player
- Sequentially would take 127 days
  - Player 1 vs player 2, its winner vs player 3, that winner vs player 4, …
  - Assuming a person can only play one match per day
- With enough courts, this takes $\log_2(128) = 7$ days
- In reality, takes 15 days as the first rounds take multiple days

Associativity

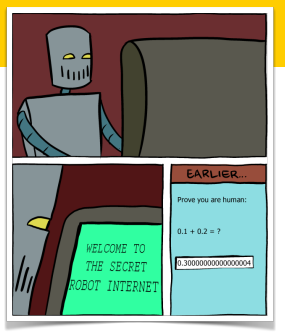
- Sum works in parallel because addition is associative
  - Sequential: $(((x + y) + z) + w)$
  - Recursive: $((x + y) + (z + w))$
- Associative: change the position of the parentheses: $((x + y) + z) \equiv (x + (y + z))$
- Commutative: change the position of the variables: $x + y \equiv y + x$
  - Example:
    - Function composition is associative: $(f \cdot g) \cdot h \equiv f \cdot (g \cdot h)$
    - But not commutative: $(f \cdot g) \not\equiv (g \cdot f)$

Associativity

- "Best" in sports is probably not associative (nor deterministic)
- Strictly speaking, computer arithmetic is not associative
  - Integer arithmetic can over/underflow
  - Floating-point values have limited precision
  - Example: 7-*digit* mantissa

```
1234.567
  45.67844
   0.000400
```

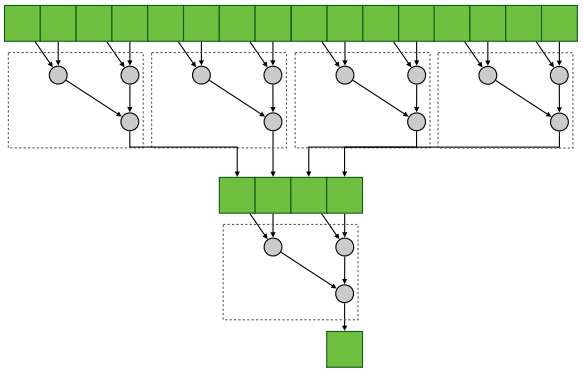
```
1234.567 + 45.67844 = 1280.24544
         + 0.000400 = 1280.2454
                    = 1280.245

45.67844 + 0.000400 = 45.67884
         + 1234.567 = 1280.24584
                    = 1280.256
```



http://www.smbc-comics.com/comic/2013-06-05
https://en.wikipedia.org/wiki/Kahan_summation_algorithm

Fold

- In practice, the input is split into multiple tiles (chunks)
- The tiles are distributed over the available cores (for CPUs) or streaming multiprocessors (GPUs)
- The results per tile are then reduced
  - With a sequential fold, or recursively with a parallel fold



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

Fold

- Reduction happens on multiple levels in the hardware
- For a GPU:
  - Each thread handles multiple elements, with a sequential loop
  - Each warp reduces the values of its threads
  - Each thread block reduces the values of its warps and writes the results to global memory
  - In a separate kernel, we reduce the results of all thread blocks
- For a CPU:
  - Each SIMD lane …
  - Each thread …
  - Afterwards, reduce the results of all threads

Example: dot product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

- The vector dot-product operation pair-wise multiplies the elements of two vectors, and then sums the result
  - A combination of `zipWith` followed by a `fold`
  - These operations can be *fused* to avoid storing the intermediate result
  - Array fusion is an important optimisation for collection-based programming models (c.f. loop fusion)

Scan

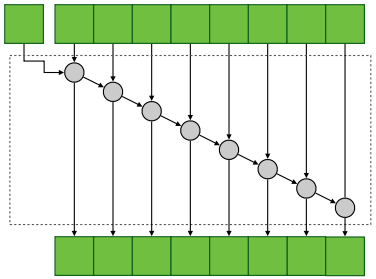
- Similar to reduce, but produces all partial reductions of the input
  - An important building-block in many parallel algorithms
    - Sorting algorithms, lexical comparison of strings, lexical analysis (parsing), evaluating polynomials, adding multi-precision numbers…
  - Trickier to parallelise than reduce
  - Two (main) variants: inclusive and exclusive
- Scan is an important building block in many parallel algorithms

https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:35

Scan

- Two variants: inclusive and exclusive
  - Inclusive scan includes the current element in the partial reduction
  - Exclusive scan includes all prior elements

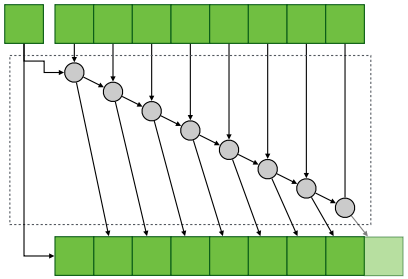
```
// inclusive: scanl1
r = initial_value;
for (i = 0; i < n; ++i) {
  r    = combine(r, x[i]);
  y[i] = r;
}
```



Scan

- Two variants: inclusive and exclusive
  - Inclusive scan includes the current element in the partial reduction
  - Exclusive scan includes all prior elements

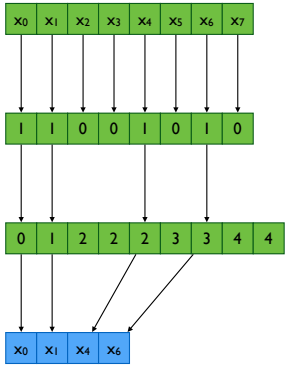
```
// exclusive: scanl
r = initial_value;
for (i = 0; i < n; ++i) {
  y[i] = r;
  r    = combine(r, x[i]);
}
// optionally: y[i] = r;
```



Example: filter (compact)

- Return only those elements of the array which pass a predicate

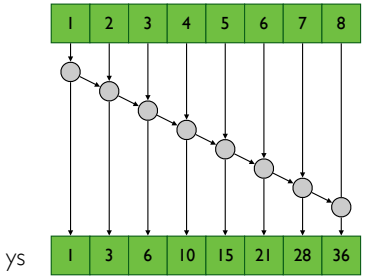
  1. *map* the predicate function over the values to determine which to keep
  2. exclusive *scan* the boolean flags to determine the output locations and number of elements to keep
  3. *permute* the values into the position given by (2) if (1) is true



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:31

Example: Integral Image

- Consider this inclusive prefix sum
  - We can use this result to calculate the sum of any interval of the input:
    `sum [3..6] = ys[5] - ys[1] = 21 - 3 = 18`



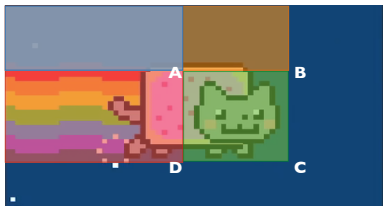
Example: Integral Image

- This idea extends to two (or more) dimensions
  - Known as the integral image or summed area table

$$I(x,y) = \sum_{v=0}^{y}\sum_{u=0}^{x} i(u,v)$$

  - Suppose I want to find the sum of the green region:

$$I_{ABCD} = I_C - I_D - I_B + I_A$$



  - Can be used to implement a box filter in constant time
  - Key component of the Viola-Jones face recognition algorithm

https://youtu.be/uEJ71VlUmMQ

Scan

- In the prefix sum we produce all partial reductions of the input
  - That is, the reduction of every prefix

```
                input = [3,4, 4, 4, 4, 3, 5, 4, 5]
scanl1 (+) input = [3,7,11,15,19,22,27,31,36]
```

  - The prefix sum you might also think of as a cumulative sum
  - Variations for inclusive, exclusive, left, right, product, conjunction…
  - Sequential calculation is a single sweep of n-1 additions

```
for (i = 1, i < n; ++i)
    A[i] = A[i] + A[i-1]
```

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

Scan

- Example: how to parallelise prefix sum

```
input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

  - Split the data over two processors and perform a prefix sum individually on each part:

```
split: [3,4, 4, 4, 4] | [3,5, 4, 5]
left/right result: [3,7,11,15,19] | [3,8,12,17]
                     P1              P2
```

  - The left part looks correct, but every element in the right part needs to be incremented by 19
  - Luckily, this is the final result of the left side, which we just computed!

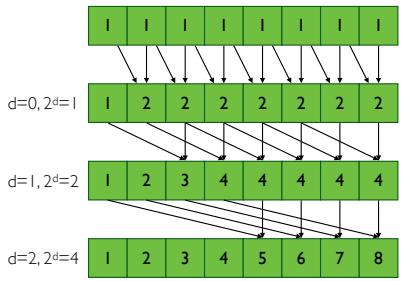
Scan

- Parallel scan split into tiles is classically done in three phases:
  1. Upsweep: Break the input into equally sized tiles, and reduce each tile
  2. Perform an exclusive scan of the reduction values
  3. Downsweep: Perform a scan of each tile, using the per-tile carry-in values computed in step 2 as the initial value

Scan

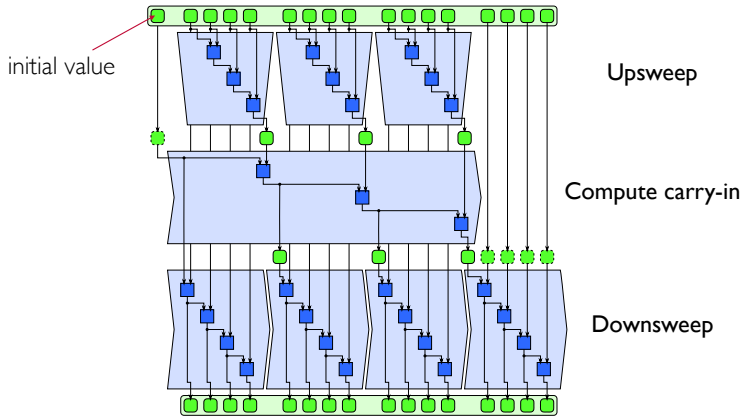
- Example: how to parallelise prefix sum (per-tile)
  - Here computed in SIMD (e.g. in a warp on the GPU)
  - Parallel scan [again] changes the order of operations

```
for ( d = 0
    ; d < log2 N;
    ; d ++ )
{
  int offset = 2^d;
  if (i ≥ offset) // parallel
    x[i] = x[i-offset] + x[i];
}
```

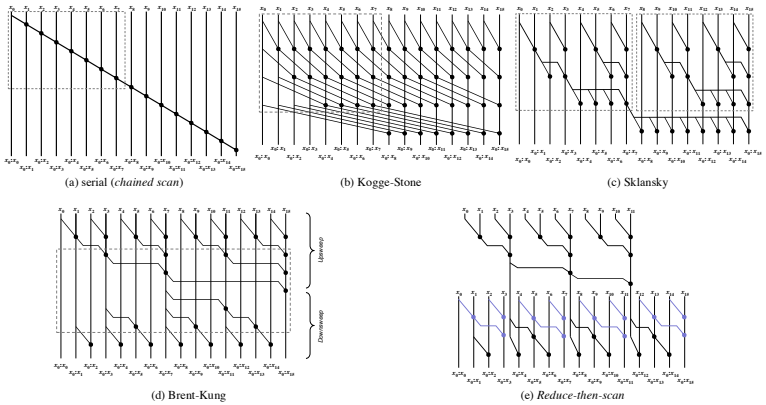


Scan

- Three-phase tiled implementation of inclusive scan:



Scan



Single-pass Parallel Prefix Scan with Decoupled Look-back, D. Merrill and M. Garland, 2016

Three-phase scans on GPUs

- Scans are (or used to be) implemented via three phases on GPUs
  - Kernel 1 performs a fold per block
  - Kernel 2 scans over the results per block (using a single thread block)
  - Kernel 3 performs a scan per block, using the prefix of that block computed in kernel 2
- Synchronization between blocks happens by splitting the program in multiple kernels
  - Kernel 2 only starts when all thread blocks of kernel 1 have finished
- It is advised to not perform synchronization between thread blocks within the same kernel
  - But…

Chained scans on GPUs

- Chained scans use only one kernel, and do synchronize within the kernel
  - Each thread block does the following:
    - Read a tile of the array
    - Fold
    - Wait on prefix of previous tile
    - Share own prefix
    - Scan
  - Three-phase scans typically split the input in a fixed number of blocks, chained scans use fixed-size blocks as the data should fit in the registers of the threads of a thread block.

https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

Chained scans on GPUs

- Chained scans go against the advice of independent thread blocks
- You have to be careful:
  - Don't use the hardware scheduler - implement your own scheduling of thread blocks
  - Prevent memory reordering
  - Waiting on the prefix of the previous block can be a significant bottleneck
    - The *Single-pass Parallel Prefix Scan with Decoupled Look-back* optimizes this
- Chained may be faster than three-phase scans
  - as they read the input once instead of twice

https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

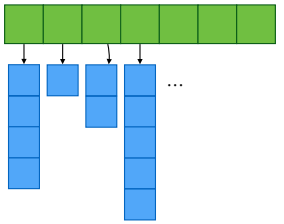
Flat data parallelism

- Widely used, well understood & supported approach to massive parallelism
  - Single point of concurrency
  - Easy to implement
  - Good cost model (work & span)
  - BUT! the "something" has to be sequential

```
__global__ void kernel( float* xs, float* ys, int n, ... )
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      // do something sequentially
      // but can not launch further parallel work!
   }
}
```

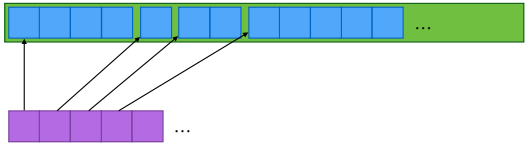

Nested data parallelism

- Main idea: allow the "something" to also be parallel
  - Now the parallelism structure is recursive and unbalanced
  - Still a good cost model
  - Wider range of applications: sparse arrays, adaptive methods (Barnes-Hut), divide and conquer (quicksort, quickhull), graph algorithms (shortest path, spanning tree)



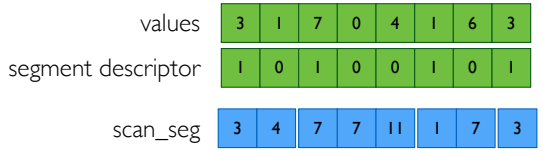
Nested data parallelism

- The flattening transformation
  - Concatenate the subarrays into one big flat array
  - Operate in parallel on the big array
  - A *segment descriptor* keeps track of where the sub-arrays begin
- Example: given an array of nodes in a graph, compute an array of their neighbors
  - For instance in findRequests for Delta-stepping
- The `scan` operation gives us a way to do this



Segmented scan

- We can also create *segmented* versions of collective operations like scan
  - Generalises scan to perform separate parallel scans on arbitrary contiguous partitions (segments) of the input vector
  - In particular useful for sparse and irregular computations



  - Can be implemented via operator transform:

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y,\ \text{if } f_y \text{ then } y \text{ else } x \oplus y)$$

Segmented scan

- Lift a binary operator to a segmented version:
  - Can be implemented via operator transform
  - The lifted operator should be associative!
    - Concretely, if $\oplus$ is associative, then $\oplus^s$ should also be associative

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y,\ \text{if } f_y \text{ then } y \text{ else } x \oplus y)$$

```
segmented
    :: Elt a
    ⇒ (Exp a → Exp a → Exp a)
    → (Exp (Bool, a) → Exp (Bool, a) → Exp (Bool, a))
segmented op (T2 fx x) (T2 fy y)
  = T2 ( fx || fy )
       ( fy ? (y, op x y) )
```


Segment descriptors

- Segment descriptors *describe* where *segments* start, via
  - Segment lengths, or
  - Head flags
- Create the *head flags* array from segment lengths
  - The segment descriptor tells us the length of each segment
  - To use the operator from the previous slide, we need to convert this into a representation the same size as the input, with a `True` value at the start of each segment and `False` otherwise

```
mkHeadFlags :: Acc (Vector Int) → Acc (Vector Bool)
mkHeadFlags seg =
  let
      T2 offset len = scanl' (+) 0 seg
      falses        = fill (I1 (the len)) False_
      trues         = fill (shape seg)    True_
  in permute const falses
       (\ix → Just_ (I1 (offset!ix))) trues
```


Segmented scan

- What about other flavours of scan?
  - This approach works directly for *inclusive* segmented scan
  - The exclusive version is similar, but needs to fill in the initial element and take care of (multiple consecutive) empty segments

Conclusion

- Fold (reduction) and scan (prefix sum) can be executed in parallel
  - if the operator is associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Prefix sum is a useful application in many (parallel) programming problems
  - Use to compute the book-keeping information required to execute nested data-parallel algorithms on flat data-parallel hardware (e.g. GPUs)


tot ziens
Photo by Anusha Barwa

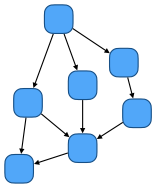
Utrecht University

B3CC: Concurrency

15: Work & Span

Ivo Gabe de Wolff

Previously...



Task parallelism

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

Data parallelism

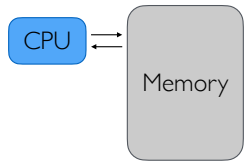
- Operate simultaneously on bulk data
- Implicit synchronisation
- Massive parallelism
- Easy to program

Performance analysis

- We want to analyse the cost of a parallel algorithm
  - We will consider *asymptotic* costs, to compare algorithms in terms of:
    - How they scale to larger inputs
    - How they scale (parallelise) over more cores
  - Example: some sorting algorithms are $O(n \log n)$ and others $O(n^2)$ over the size of the input
  - Example: RTX 4090 Ti has 16384 "cores" distributed over 128 multiprocessors

RAM

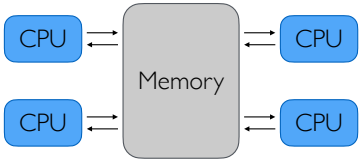
- When designing and analysing sequential algorithms, we use the random access machine (RAM) model
  - All locations in memory can be read from & written to in $O(1)$
  - Summing an array can be done in linear $\Theta(n)$ time




```
s = 0
for (i = 0..n)
  s := s + arr[i]
```

PRAM

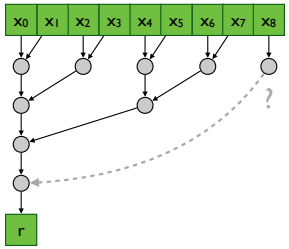
- The parallel random access machine (PRAM) model is analogous for talking about parallel algorithms
  - Shared memory machine with multiple attached processors (cores)
  - Ignore details of synchronisation, communication, etc.
  - Question: can we sum an array in parallel using this algorithm?




```
s = 0
parallel_for (i = 0..n)
  s := s + arr[i]
```


Fold

- Binary tree reduction of an array
  1. For even `i`:
     `arr[i] += arr[i+1]`
  2. For `i` a multiple of 4:
     `arr[i] += arr[i+2]`
  3. For `i` a multiple of 8:
     `arr[i] += arr[i+4]`
  4. et cetera…



Fold

- Binary tree reduction of an array
  - To calculate step one instantly you need $n/2$ processors: $O(n)$ operations and the whole algorithm takes $O(\log n)$ time
  - The *hardware cost* is thus the number of processor P multiplied by how long you need them: $O(n \log n)$
  - So, we can go faster with parallelism but at a higher hardware cost. Can this be improved?
    1. Can we go faster than $O(\log n)$?
    2. Can we have less hardware cost than $O(n \log n)$?

Fold

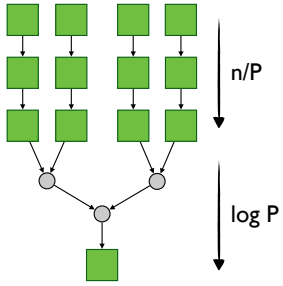
- Question 1: can we sum an array in sub-logarithmic time?
  - Addition is a binary operator
  - Parallel execution of binary operators can, after i rounds, produce values that depend on at most 2^i values
  - So, no matter what you do in parallel, you can not compute the full sum of n numbers in less than $O(\log n)$ time

Fold

- Question 1: proof by induction
  - Induction hypothesis (IH): after i rounds values can only depend on at most 2^i inputs
  - i=0: After zero rounds we haven't done anything, so a number only "depends" on itself, so on one number which is 2^0
  - $i+1$: In this round you can combine two inputs from round i, which according to the IH can only depend on at most $2^i + 2^i = 2^{(i+1)}$ inputs
  - Therefore, addition can not be done sub-logarithmically. This holds true for all binary operators, which is why (poly)logarithmic complexity $O(\log^c n)$ is the best possible outcome for parallel execution

Fold

- Question 2: can we reduce the hardware cost?
  - Split the problem into two steps
  - Phase 1: divide the input over the P processors in groups of length n/P
  - Phase 2: use a binary tree reduction to calculate the total from the P partial sums
  - Total time $T_p = n/p + \log p$
    - If $P \leq n / \log n$ then phase one is dominant
    - If $P \leq n / \log n$ then hardware cost is $O(n)$

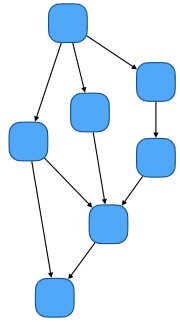


Work & Span

- We don't want a different optimal calculation when executing for a different number of cores
  - Use a description with two parameters, instead of just sequential time
  - Let T_p be the running time with P processors available
  - Then calculate two extremes: the *work* and *span*
- **Work = T_1:** How long to execute on a single processor
- **Span = T_∞:** How long to execute on an infinite number of processors
  - The longest dependence chain / critical path length / computational depth
  - Example: $O(\log n)$ for summing an array

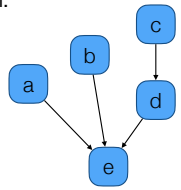
Work & Span

- Program can be seen as a dependency graph of the calculation steps
  - *Work* is the total number of nodes (calculations) in the whole graph
  - *Span* is the number of nodes on the longest path (height of the graph)



Work & Span

- If the work and span are known, you can estimate the time on P processors T_P with:
  - $\max(work/P, span) \leq T_P \leq work/P + span$
  - The latter is at most double the former, so:
    - $T_P = O(work/P + span)$
  - Question: what is the time to execute on 1, 2, or 3 cores?



Scheduling

- Brent proved that *greedy scheduling* is always two-optimal
  - We say a step is ready when all its predecessors (dependencies) have been computed in previous rounds
  - A greedy scheduler does as many steps in a round, but does not care which
  - This is two-optimal:
    Greedy scheduling takes at most twice as long as the optimal schedule
- Say T_P^* is the time for the optimal schedule, then:
  - $T_P^* \geq work/P$, because even the best schedule still has P cores available
  - $T_P^* \geq span$, because all calculations on a path must be done sequentially

Scheduling

- Greedy scheduling
  - Full round: if there are P or more steps ready, do P steps this round; this happens at most $work/P$ times
  - Empty round: there are fewer than P steps ready; this happens at most $span$ times, because every round the span decreases by one
  - The length of the greedy schedule is:

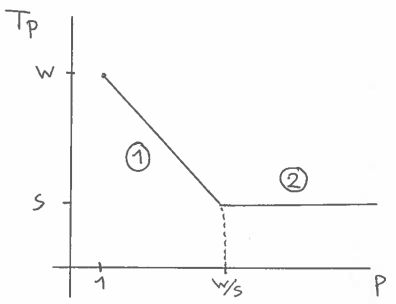
$$\begin{aligned} T_P &= full + empty \\ &\leq work/P + span \\ &\leq T_P^* + T_P^* \\ &\leq 2T_P^* \end{aligned}$$

Scheduling

- Greedy scheduling
  - Greedy scheduling has length at most twice the length of the optimal one, so is asymptotically optimal
  - Because $work/P + span$ and $max(work/P, span)$ are asymptotically equal (differ by a factor of two), we can say that $T_P = max(work/P, span)$

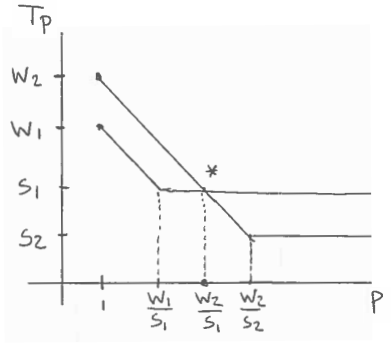
Work & Span

- Greedy scheduling
  1. As long as $P \leq work/span$ the first term is dominant and the calculation can be shortened by adding more cores: *work bound phase*
  2. If we have $P > work/span$ then the runtime will not get shorter by adding more cores: *span bound phase*



Work & Span

- When comparing algorithms, low work is better than high work, and low span is better than high span
  - What if algorithm one has better work complexity $w_1 < w_2$
  - But algorithm 2 has better span complexity $s_1 > s_2$
  - Low span is theoretically nice, but since we don't have infinite processors in practice, be careful not to lower span at the cost of too much extra work



Work & Span

- Calculating work and span is the same as computing the time of an algorithm, as learned in the course data structures
  - Count the number of instructions/operations
  - In the case of a loop, the cost of the body times the number of repetitions
  - For recursion, use the Master Theorem
- For the analysis of parallel algorithms:
  - You must do this process twice, once each for work and span
    - Work is done as you would for a sequential algorithm
    - Span takes the maximum of the branches which are performed in parallel

Example: zipWith

- Pair-wise multiply the elements of two arrays

```
1   parallel_for (i = 0..n)
2     r[i] := x[i] * y[i]
```

- Work analysis:
  - Doesn't care about parallelism
  - Line one says that this is done n times, so costs $\Theta(n)$ steps
- Span analysis:
  - The maximum cost of all the branches which are done in parallel
  - Loop on line 1 is parallel, so take the longest path of steps: $\Theta(1)$

Example: fold (1)

- Add up all the numbers in an $n \times n$ matrix A, with subtotals per row

```
1   parallel_for (j = 0..n)
2     s[j] = 0
3     for (i = 0..n)
4       s[j] := s[j] + A[i,j]
5   t = 0
6   for (i = 0..n)
7     t := t + s[j]
```

- Work analysis:
  - Loop on line 3-4 costs $\Theta(n)$ steps
  - Line one says this will be done n times, so line 1-4 take $\Theta(n^2)$ steps
  - Line 6-7 take $\Theta(n)$ steps
  - Total is $\Theta(n^2)$ work

Example: fold (1)

- Add up all the numbers in an $n \times n$ matrix A, with subtotals per row

```
1   parallel_for (j = 0..n)
2     s[j] = 0
3     for (i = 0..n)
4       s[j] := s[j] + A[i,j]
5   t = 0
6   for (i = 0..n)
7     t := t + s[j]
```

- Span analysis:
  - Loop line 3-4 is sequential, $\Theta(n)$ steps
  - Loop line 1 is parallel, so we take the longest path of steps from line 1-4: $\Theta(n)$
  - Line 5-7 still have $\Theta(n)$ sequential steps
  - Total span is $\Theta(n)$ steps

Example: fold (2)

- Parallel algorithms can often use recursion effectively
  - We want a method `sum(A, p, q)` that calculates the sum of all numbers in A in the range $[p,q)$
  - Using recursion, pretend you already have a clever way to sum $n/2$ numbers, which you want to use to calculate the sum of n numbers

```
1   sum (A, p, q)
2     parallel_for (i = 0..(q-p)/2)
3       B[i] = A[p+2*i] + A[p+2*i+1]
4
5     sum (B, 0, (q-p)/2)
```

  - Ignore possibility of uneven number of inputs, base case of recursion, etc…

Master Theorem

- The master theorem provides a solution to recurrence relations of the form
  - For constants $a \geq 1$ and $b > 1$ and f asymptotically positive

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- The master theorem has three cases:

Recursion dominates	Both contribute	*f* dominates
If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$	If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$	If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$, and $af\left(n/b\right) \leq cf\left(n\right)$ for some $c < 1$ for all n sufficiently large, then $T(n) = \Theta\left(f\left(n\right)\right)$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Recall: Master Theorem

- The master theorem provides a solution to recurrence relations of the form
  - For constants $a \geq 1$ and $b > 1$ and f asymptotically positive

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Examples:
  - Merge sort: $T(n) = 2T(n/2) + n$
    Then case 2 gives (a=2, b=2): $T(n) = \Theta(n \log n)$
  - Traversing a binary tree: $T(n) = 2T(n/2) + O(1)$
    Then case 1 gives (a=2, b=2, ε=1): $T(n) = \Theta(n)$

Example: fold (2)

- Parallel algorithms can often use recursion effectively

```
1   sum (A, p, q)
2     parallel_for (i = 0..(q-p)/2)
3       B[i] = A[p+2*i] + A[p+2*i+1]
4
5     sum (B, 0, (q-p)/2)
```

- Work analysis:
  - Line 3 is $\Theta(1)$
  - Line 2 says it is done $n/2$ times, so $\Theta(n/2)$
  - Line 3 is a recursive call on $n/2$ inputs. Call the work $W(n)$ and we get $W(n) = W(n/2) + \Theta(n/2)$
  - Solve with the master theorem (a=1, b=2, ε=1, case 3): $W(n) = \Theta(n)$

Example: fold (2)

- Parallel algorithms can often use recursion effectively

```
1   sum (A, p, q)
2     parallel_for (i = 0..(q-p)/2)
3       B[i] = A[p+2*i] + A[p+2*i+1]
4
5     sum (B, 0, (q-p)/2)
```

- Span analysis:
  - Line 2-3 have constant span because they are done in parallel
  - This means the span $S(n) = S(n/2) + \Theta(1)$
  - Solve with the master theorem (a=1, b=2, case 2): $S(n) = \Theta(\log n)$
- *Conclusion*: we can sum n numbers in linear work and logarithmic span

Example: scan (1)

- Parallel implementation of prefix sum

```
input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

  - Split the data over two processors and perform a prefix sum individually on each part

```
split: [3,4, 4, 4, 4] | [3,5, 4, 5]
left/right result: [3,7,11,15,19] | [3,8,12,17]
                    P1                 P2
```

Example: scan (1)

- Example: recursive implementation of prefix sum:

```
1  prefix_sum (A, p, q)
2    // base case
3
4    m = (p+q)/2
5    prefix_sum(A, p,   m)   | In parallel
6    prefix_sum(A, m+1, q)   |
7    parallel_for (i = m+1..q)
8      A[i] = A[i] + A[m]
```

- Span (a=1, b=2, case 2): $S(n) = S(n/2) + 1 = \Theta(\log n)$
- Work (a=2, b=2, case 2): $W(n) = 2\ W(n/2) + n = \Theta(n \log n)$

Efficient & optimal

- The parallelisation *overhead* of an algorithm is its work divided by the cost of the best sequential algorithm
  - For this parallel scan we have to put $O(n \log n)$ work into something which can be done sequentially in linear $O(n)$ time: the overhead is logarithmic
  - A parallel algorithm is:
    - *Efficient* when the span is poly-logarithmic and the overhead is also poly-logarithmic
    - *Optimal* when the span is poly-logarithmic and the overhead is constant

Example: scan (2)

- Let's try a different approach to parallelising scan:

```
input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

- Pair up neighbours at the even positions:

```
[  7,    8,    7,    9   ]
```

- Perform a prefix sum of these values:

```
[  7,   15,   22,   31   ]
```

- At the uneven positions add the input value at that position to the output of the previous step on the left:

```
[3,7,11,15,19,22,27,31,36]
```


Example: scan (2)

- We can implement this recursively by keeping track of a hop distance

```
1  prefix_sum (A, d)
2    parallel_for (i = even multiple of d)
3      A[i] += A[i-d]
4    prefix_sum(A, 2*d)
5    parallel_for (i = uneven multiple of d)
6      A[i] += A[i-d]
```

- Work:
  - Algorithm does n-1 additions and one half-size prefix sum
  - Master theorem (a=1, b=2, ε=1, case 3): $W(n) = W(n/2) + n = \Theta(n)$

Example: scan (2)

- We can implement this recursively by keeping track of a hop distance

```
1  prefix_sum (A, d)
2    parallel_for (i = even multiple of d)
3      A[i] += A[i-d]
4    prefix_sum(A, 2*d)
5    parallel_for (i = uneven multiple of d)
6      A[i] += A[i-d]
```

- Span:
  - Additions are done in two (parallel) groups, before and after the prefix sum
  - Master theorem (a=1, b=2, case 2): $S(n) = 1 + S(n/2) + 1 = \Theta(\log n)$
  - Since the span is logarithmic and there is no overhead, this prefix sum is parallelised optimally

Summary

- Work and span are used to analyse and compare asymptotic behaviour of parallel algorithms
  - Work: total number of steps (computations)
  - Span: longest path of steps that need to be done sequentially (steps)
- The PRAM model ignores practical issues such as memory access latency
  - Assume uniform costs for all memory access
- Time to perform something on P cores: $T_P = \Theta(work/P + span)$
  - Compare to the formulation by Amdhal

Next time...

- Thursday: Revision lecture
  - This will consist of the last lectures presented simultaneously (it is up to you to parallelise your brain before then)
  - *Send me questions/topics to cover via Teams!*

Photo by Claudio Piccolo

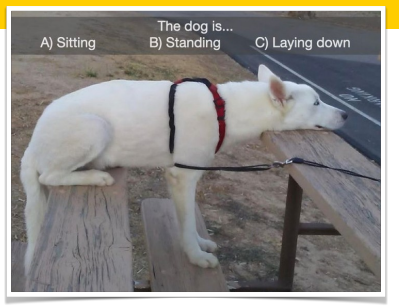
Utrecht University

B3CC: Concurrency

16: Conclusion

Ivo Gabe de Wolff

Final exam!



- Final exam
  - Tuesday 30 January @ 13:30
  - Olympos Hal 2
  - Mix of multiple choice and open questions
  - Covers material from second half of the course:
    - From lecture 9 (Parallelism) through lecture 15 (Work & Span)
  - You don't need to write Accelerate code
    - But you may be asked to design a parallel algorithm in terms of the parallel patterns
  - Remindo has a calculator, no physical calculators allowed

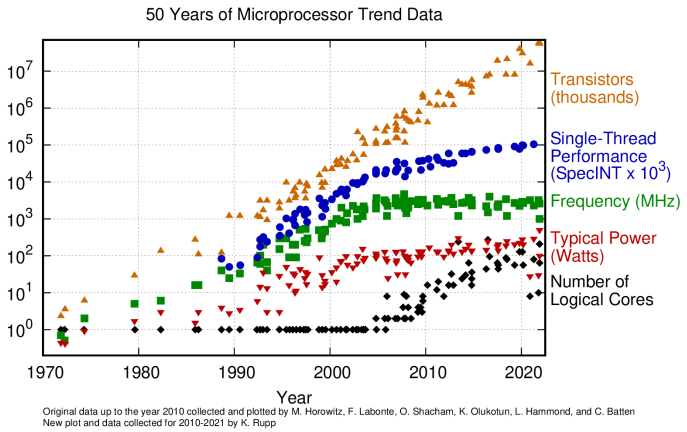
https://www.instagram.com/p/CZIPjkjjxj1

What?

Brief course summary

Parallelism & Concurrency

Why?



https://github.com/karlrupp/microprocessor-trend-data

Where?

- Three kinds of code:
  - Gameplay simulation
    - Models the state of the game world as interacting entities
  - Numeric computation
    - Physics, collision detection, path finding, scene graph traversal, etc.
  - Shading
    - Pixel & vertex attributes; runs on the GPU

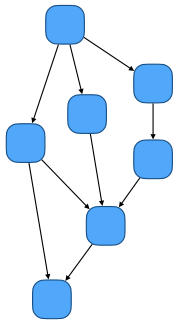
Sekiro: Shadows Die Twice, FromSoftware

How?

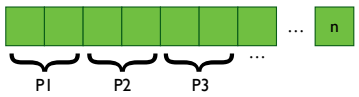
	Game Simulation	Numeric Computation	Shading
Languages	C++, scripting	C++	GC, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250.000	250.000	10.000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS
Concurrency/ Parallelism	**STM**	**SIMD**	**GPU**

Tim Sweeney: *The Next Mainstream Programming Language*, POPL 2006

Kinds of parallelism



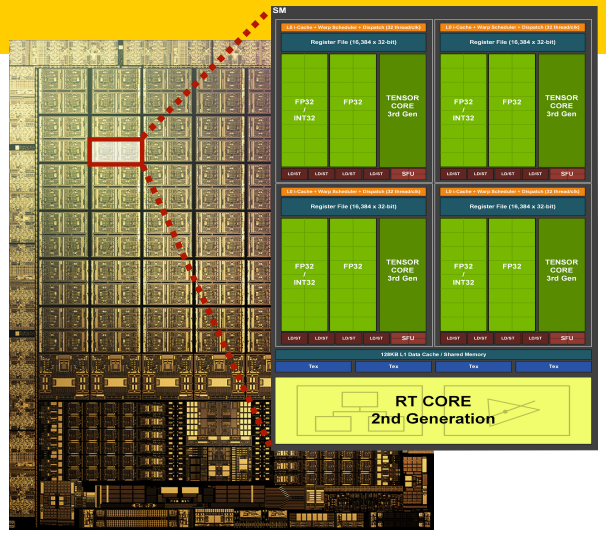
Task parallelism



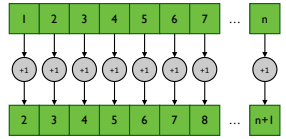
Data parallelism

GPGPU

- How the parallel patterns we have talked about map to GPU code
- Difference between CPU and GPU
  - What each is designed for; strengths and weaknesses
  - What the GPU programming model (CUDA) is designed for

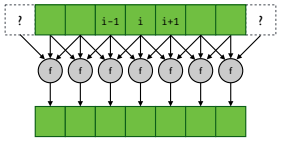


Patterns: map



- Apply a function to every element of an array, independently
- This one is (hopefully) straightforward…

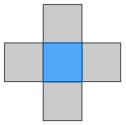
Patterns: stencil



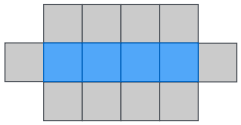
- A map with access to the surrounding neighbourhood
- What are the difficulties/limitations?
  - The ghost region (halo) and how/why to use it
  - Optimisations (tiling, strip mining, etc.)

Stencil optimisations

- Use a different kernel for the interior and border regions
  - In the gaussian blur example of a 512x512 pixel image, 98% of the pixels do not require in-bounds checks
- Optimise data locality & reuse through tiling
  - *Strip mining* is an optimisation that groups elements in a way that avoids redundant memory access and aligns accesses with cache lines



4 x (5 reads + 1 write)

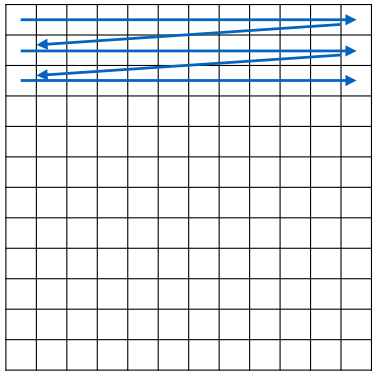


14 reads + 4 writes

Stencil optimisations

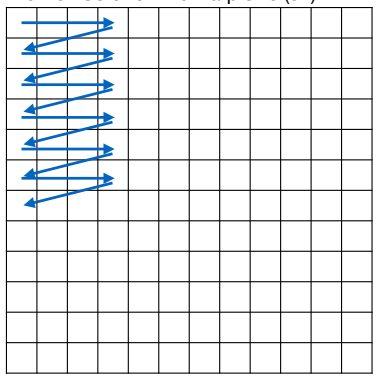
Without tiling

- When handling row 0, row 1 is loaded in cache.
- First values of row 1 may already be out of cache, when handling row 1

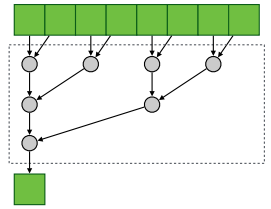


With tiling

- Previously loaded row is still in cache
- Tile width is usually a power of 2, on GPUs often the warp size (32)

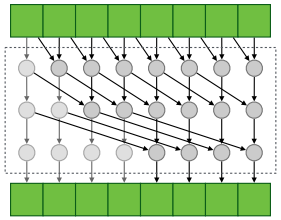


Patterns: fold



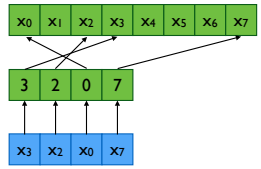
- Reduce an array to a summary value
- How to implement this in parallel
  - What kinds of restrictions are necessary?
  - What additional restrictions can be leveraged to improve it further?

Patterns: scan



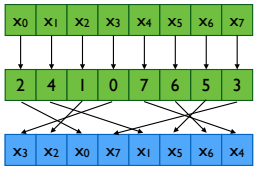
- All partial reductions of an array
- Varieties
  - Inclusive vs. exclusive etc.
- Parallel implementation
  - Restrictions, etc.

Patterns: gather



- Parallel random read
- Implications for memory access patterns
  - Optimisations for special cases (e.g. transpose), like tiling
  - Implications for the GPU, caches, etc.

Patterns: scatter



- Parallel random write
- How to handle collisions in the index permutation function
  - Performance implications of collisions, false sharing, etc.
  - Scatter vs. gather

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

Patterns: gather vs scatter

- Random reads (gather) are slower than structured reads
- Random writes (scatter) are slower than structured writes
- This problem is larger for scatter,
  as the processor needs to perform more synchronization between cores
- In general, use gather instead of scatter if both are possible

Patterns

- You should be able to:
  - Give examples for each pattern
  - Recognise these patterns and where they can be used
    - e.g. given a problem description, give an implementation in terms of these patterns
    - Use Accelerate code, pseudocode or an explanation in text
      - Especially for the latter, make sure your explanation is concrete

Work & Span

- We analysed the performance of algorithms using the *work* and *span*:
  - **Work = T_1**
    How long to execute on a single processor
  - **Span = T_∞**
    How long to execute on an infinite number of processors
    - The longest dependence chain / critical path length / computational depth
    - Example: $O(\log n)$ for summing an array

Efficient & optimal

- The parallelisation *overhead* of an algorithm is its work divided by the cost of the best sequential algorithm
  - For this parallel scan we have to put $O(n \log n)$ work into something which can be done sequentially in linear $O(n)$ time: the overhead is logarithmic
  - A parallel algorithm is:
    - *Efficient* when the span is poly-logarithmic and the overhead is also poly-logarithmic
    - *Optimal* when the span is poly-logarithmic and the overhead is constant

Master Theorem

- The master theorem provides a solution to recurrence relations of the form
  - For constants $a \geq 1$ and $b > 1$ and f asymptotically positive

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- The master theorem has three cases:

Recursion dominates	Both contribute	*f* dominates
If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$	If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$	If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$, and $af\left(n/b\right) \leq cf\left(n\right)$ for some $c < 1$ for all n sufficiently large, then $T(n) = \Theta\left(f\left(n\right)\right)$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Analysis of parallel algorithms

- You should be able to:
  - Compute the work and span given a problem description/code
  - Compare parallel algorithms
    - Efficient & optimal
    - Parallel speedup (Amdhal vs. Gustafson-Baris)

Questions?

Finally…

- Please fill out the Thermometer survey!
  - All constructive feedback is welcome
  - https://caracal.uu.nl/35916/Respond

Photo by Claudio Piccolo