The perfectly parallel program

Architectures for hardware acceleration and heterogeneous computing

Zenny Wettersten (CERN, TU Wien)

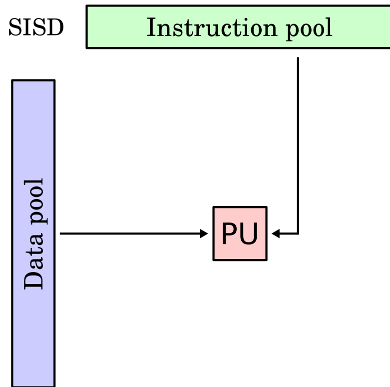
Structure

- Two hour-long lectures
  1. Technical overview of processors in different hardware architectures
     - What is a processing unit, an instruction fetch, a data fetch?
     - How do we classify the architectures available to us today?
     - Can data parallelism solve bottlenecks in modern scientific computing?
     - Which restrictions do accelerators put on our software?
  2. Design philosophy for (data-level) parallel software
     - How do we build into the structure of accelerators?
     - What makes a program "parallel"?
     - How do we design an algorithm in mind?
     - Can we redesign legacy code to make use of hardware acceleration?
- One exercise session where we put this into practice

Processing units

Single instruction, single data stream

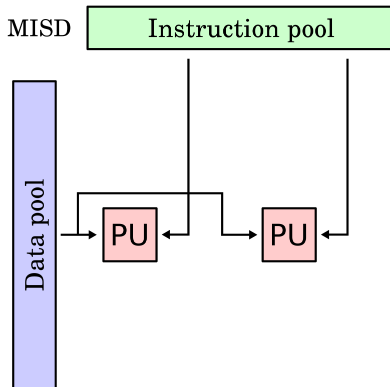
- Standard processing unit
- Takes one operation and one piece of memory
  (One instr. fetch, one data fetch)
- Performs operation on the datum
- Outputs modified datum in memory
- Examples: Single-threaded CPU
- Sequential in both data and instr.
  Can be parallelised along two axes



(Cburnett, CC BY-SA 3.0)

Multiple instructions, single data stream

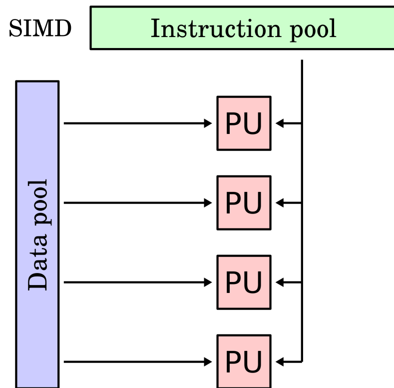
- Parallelise along operations, i.e. several instr., same datum
- Hypothetically good for complicated problems on small parameter spaces
- Problematic; not seen in practice
- Digital CPU instructions are typically not commutative, simultaneous application undefined
- Will not be elaborated on here



(Cburnett, CC BY-SA 3.0)

Single instruction, multiple data

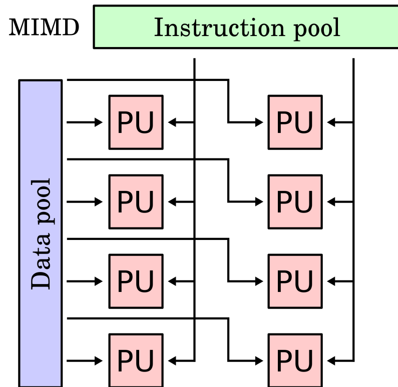
- Parallelise along data, i.e. same instr., many data
- Good when the same numerical routines are run on many data points
- No additional requirements besides memory structure itself
- Usually what we mean by *"parallel hardware"*
- Examples: GPUs, vector CPUs



(Cburnett, CC BY-SA 3.0)

Multiple instructions, multiple data

- Parallelise along both axes, i.e. have separate PUs running distinct tasks simultaneously
- Allows running many processes on the same machine at once
- Overwhelmingly common nowadays
- Examples: Multi-core CPUs, clusters



(Cburnett, CC BY-SA 3.0)

Relevant types of parallelism

- Task-level parallelism
  - Running separate parts of a program across multiple nodes (MIMD)
  - E.g. running I/O, analysis, plotting as separate simultaneous instances
  - Comparatively simple to implement, fewer considerations than SIMD
  - Need to ensure data integrity for shared memory
  - Only a sidenote in these lectures
- Data-level parallelism
  - Applying same operations across many data in parallel, think "flipping a `for each` loop horizontally" (SIMD)
  - E.g. changing brightness of pixels, translating points in a coord. system
  - Requires strict independence between "iterations"
  - Need to ensure both data integrity and common workflow
  - Primary subject today

The hardware of today

- Multi-core processors, clusters
  - "Standard" machines you'll find today, you'll have access to *n* cores
  - Built with task parallelism in mind, but can do data parallelism by forking parallelisable sections across multiple threads
- Vector processors (SIMD CPUs)
  - CPUs with *vector memory registers* and access to SIMD instruction sets
  - If code is set up for it, can accelerate program just with compiler flags
  - Very common nowadays even among consumer grade CPUs (my laptop has support for AVX-512, can operate on 64 bytes at once)
- Graphics processors (GPUs)
  - Specialised hardware for parallel problems with many parallel threads
  - Single instruction across multiple threads (SIMT)

Defining acceleration

Question 0

We have a program which performs an operation `OP` on some `DATA`. Assume that I/O takes exactly 2 s, and it takes 0.5 s to perform `OP` on one unit of `DATA`.
For 16 units of `DATA`, the completely sequential runtime is 10 s. However, our machine has a SIMD CPU with a 16-fold vector register[1]. What speedup does our SIMD CPU give us?

[1] It can perform `OP` on 16 units of `DATA` in the same clock cycle.

Answer

The question is ill-defined.
What do we actually mean by speedup?

Software scalability

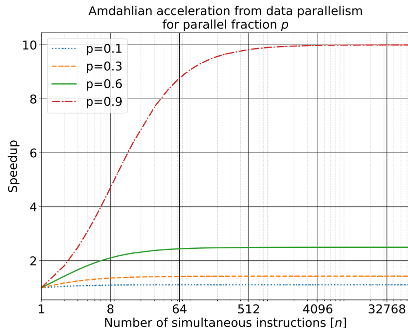
- Need to define what acceleration *is* in our context,

$$\text{“True” speedup} = \frac{\text{data}_{par}}{t_{par}} \Bigg/ \frac{\text{data}_{seq}}{t_{seq}}$$

- Are we keeping the amount of data or the runtime constant?
- Two options:
  - How much faster does our program run on the same data?
  - How much more data can our program run on in the same runtime?

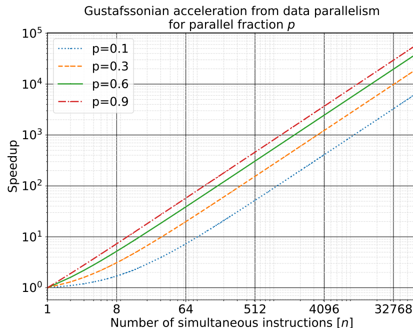
Hard and soft scaling, given parallel runtime fraction p

Ahmdahl's law:



$$\text{Speedup} = \left.\frac{t_{par}}{t_{seq}}\right|_{\text{data}}$$

Gustafsson's law



$$\text{Speedup} = \left.\frac{\text{data}_{par}}{\text{data}_{seq}}\right|_{t}$$

Computing budgets

- Which definition to use depends on what our bottleneck is
  - If we have a set problem with pre-defined data, consider Amdahl
  - If we have a set time[2] to use as we please, consider Gustafsson
- For HEP purposes (and even HPC in general):
  - At HPC centres (including WLCG), users are allotted x CPU hours
  - Moving into the HiLumi era, we need *more data points*
- $\rightarrow$ For most purposes, Gustafsson is an appropriate metric

[2] This could be either CPU time or wall time. For SIMT machines, acceleration is typically only with respect to wall time, but for SIMD machines we accelerate both with respect to CPU and wall time.

Hardware limitations for data parallelism

Question 1

How many threads does a modern HPC-grade GP-GPU have?

Answer

Manufacturer	Model	No. parallel PUs
NVidia	A100	6 912
	H100	14 592
Intel	Flex 140	2 048
	Flex 170	4 096
AMD	MI 250X	14 080
	MI 300X	19 456

The difference between SIMD and SIMT

- SIMD CPUs work by having one PU act on a vector register
  - Limited by register size — can only have so much data at the same spot
- GPUs instead set up many PUs to act synchronously
  - Memory split up between threads, limiting factor instead threads
  - Architecture is simplified by synchronous threads (SIMT)
- SIMD architectures act on many data at once,
  SIMT architectures have many threads acting identically[3]

[3] Note: Depending on the architecture, SIMT threads might also support SIMD instructions within a single thread.

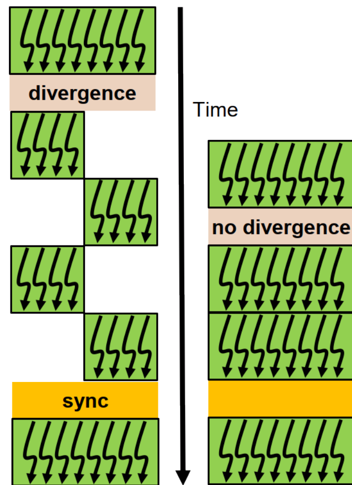
Branching code

- We've talked a lot about synchronous program flow
  - The issue arises when we have conditional expressions;
    `if` statements, `while` loops, `cases`
- By definition, SIMD data parallelism breaks down at conditionals
  - Modern SIMD CPU registers hold up to 512 bits
    (16 single precision or 8 double precision floating point numbers);
    not unreasonable to ensure 8-16 data points follow same program flow
- What about SIMT hardware?
  - As mentioned, modern GP-GPUs have thousands of threads
  - Ensuring next $\sim$ 10k data points take same path may sound hard
  - But SIMT doesn't (primarily) use vector registers;
    the actual PUs are independent but working synchronously!

Lockstep processing

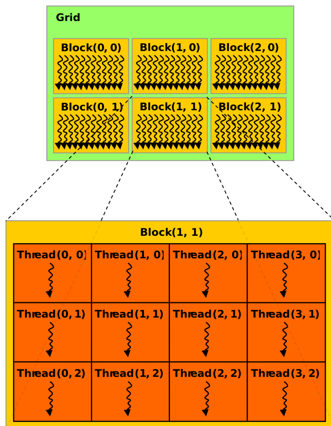
- Threads *locked* to *step* simultaneously, each thread performs *same* instruction
- But the PUs are independent; if a condition fails, the thread can wait
- For SIMT, sychronicity isn't destructive, won't perform wrong instruction on data
- *Will* limit gain from SIMT, though

⇒ *Minimise* thread divergence for SIMT, but fine not to eliminate fully



(NVidia Volta whitepaper, mod.)

Limiting the scope of lockstep



(NVidia, CC BY 3.0)

- Hardware manufacturers recognise difficulty in perfect synchronicity
- Instead of having *all* threads in lockstep, group *blocks* of *n* threads together
- Also simplifies use of shared data; blocks have shared cache instead of having all shared data in global memory
- Inter-thread communication reasonable *within a given block*

Question 2

Let's take a step back:
Assuming destructive synchronicity[4], how can we ensure that our program is safe to run data-parallel?

[4] I.e. perfect synchronicity will ruin our data in case of branching program paths. A SIMD algorithm can run on SIMT, but not vice versa.

Answer

Assuming we have full control of the code, we have a couple options:

- Pre-processing
  - Factorise conditionals to the program *top*
  - Works well when conditions are known up front and take little time
- Post-processing
  - Factorise conditionals to the *bottom*, throwing away those that fail
  - Works well when we know a priori most sampled points will be relevant
- Branch-free arithmetic (*predication*)
  - Rewrite conditionals into singular eqs
  - Useful when branches are small or logic is difficult to evaluate a priori
  - (Necessarily means evaluating all branches: Overhead)

Example of pre- and post-processing: NLO event generation

```
1: p ← genMomentum()
2: if passCuts(B, p) then
3:     M(B) ← Born(p)
4: if passCuts(R, p) then
5:     M(R) ← Real(p)
6: if passCuts(S, p) then
7:     M(S) ← Soft(p)
8: if passCuts(C, p) then
9:     M(C) ← Coll(p)
```

```
1: while !passAnyCuts(p) do
2:     p ← genMomentum()
3: M(B) ← Born(p)
4: M(R) ← Real(p)
5: M(S) ← Soft(p)
6: M(C) ← Coll(p)
7: for a ∈ [B, R, S, C] do
8:     if !passCuts(a, p) then
9:         M(a) ← 0
```

Branch-free programming

- Evaluate both branches and "multiply" by truth value
- Comes at the cost of always evaluating both branches
  - For SIMT, no time loss compared to branching
  - For SIMD, avoids destructive synchronicity
- Example from Algorithmica, summing random numbers:

```
1: for i = 0, ..., N do
2:     A(i) ← RAND()%100
3: for i = 0, ..., N do
4:     if A(i) < 50 then
5:         s ← s + A(i)
```

```
1: for i = 0, ..., N do
2:     A(i) ← RAND()%100
3: for i = 0, ..., N do
4:     s ← s + (A(i) < 50) × A(i)
```

Heterogeneous computing

Overhead from parallelism

	SIMD CPUs	Multi-core	GPUs
Launch	No	Yes	Yes
Memory	No	Yes	Yes
Processing	Yes	No	Yes

Table illustrating the overhead of different architectures. SIMD CPUs come with little to no overhead, besides the data management necessary to ensure data parallelism. On the other hand, GPUs come with a lot of overhead, illustrating the need to program with their limitations in mind.

Physical and practical distance between host and device

- Moving to heterogeneous computing,
  inter-device communication becomes a bottleneck
- The host and device are distinct: Need to keep track of data sent back and forth, and *when* that data is sent and received
- Data transfer takes time, but waiting idly wastes resources
- To optimise a parallel program, must consider both these things

Communicating with off-CPU devices

- Need to maximise off-CPU time
- Device tasks too simple and batches too small, more time spent sending data than running code
- Example: I work on event generation on GPUs
  - For very simple processes ($e^+e^- \to \mu^+\mu^-$), speedup less than a factor 2
  - For complex processes ($gg \to t\bar{t}ggg$), speedup $\sim$ a factor 80
- Synchronising host and device generally a hard bottleneck

TIME (t)

Host → Device → Host → Device → Host → Device → Host → Device → Host

Host → Device → Device → Device → Device → Host

Question 3

Assume we have 1 Gb/s connection to our device. Each data point takes 640 bits of data[5]. If we're running our program on 100 000 data points, we need to move a total of 6.4×10^7 bits to our device.

In this situation, what is the real time overhead from data transfer (assuming device launch time is negligible)?

[5] This could mean each point takes e.g. 10 double precision or 20 single precision floating point numbers.

Answer

Trick question!
The limiting factor is latency, not bandwidth.
Unless you're working with incredibly large data sets per data point, transfer time will be practically identical regardless of size.

Minimising communication, maximising concurrency

- Rule of thumb: Make the device code as large as possible
  - It may be worth keeping conditionals in GPU code, if it means minimising inter-device communication
  - Also need to make sure you're running the right code on your device: Code is never slow until it has been profiled (imagine spending time and effort porting a part of your code taking a fraction of runtime — neither Amdahl nor Gustafsson will save you)
- Also, batches need to be large enough to saturate device (e.g. CUDA now supports data streams to and from device, allowing host code to run until device data returns)

Summarising the fundamental issues

The limits of heterogeneous computing

- Accelerators like GPUs can speed up code significantly *when used properly*
  - Minimise branching for set block-sizes
  - Privatise data and minimise inter-thread communication
  - Maximise batch sizes and ensure all needed data available up front
- Heterogeneous computing is limited by communication
  - The less communication necessary, the better
  - Whether a different CPU or GPU, want to send it a large task and let it chug, while the host can do something else in the meantime

Hardware-specific limitations

1. Multi-core systems:
   - Inter-core communication expensive, needed data should be preloaded
   - Consequently, tasks are largely blind to each other:
     Inter-task dependencies will lead to time spent waiting
2. SIMD CPUs
   - Destructive synchronicity; branching within a vector register ruins data
   - Provides "free" speedup, but limited by register size[6]
3. GPUs
   - High latency, leading to long wait times
   - Lockstep processing; synchronicity non-destructive but inefficient, both due to idle time and possible random memory access
   - (For non-data centre GPUs: Bad at double precision arithmetic)[7]

[6] Today, biggest vector registers are 512 bits: 16 floats, or 8 doubles.

[7] Gaming GPUs and ML-focused devices don't have proper double precision PUs, instead simulate it with floating point precision instructions.

Data safety in multithreaded contexts

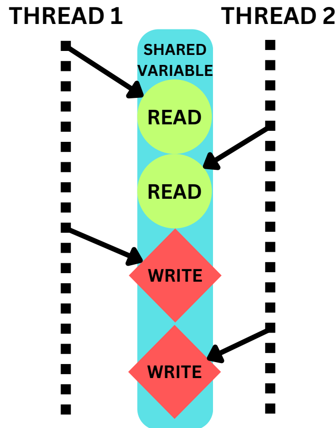
Data integrity and race conditions

- Until now, we've assumed no data is shared between threads
- Of course, not all problems are embarrassingly parallel[8]
- Introducing global data to parallelism yields *race conditions*
  - When two threads access a variable, which one comes first is a coin flip
  - The two threads are *racing* to the datum
  - If this involves write access, the result depends on order — random
- (Note: Problem for *write* access — read-only global access fine)

[8] An embarrassingly parallel problem is one where iterations are completely independent from each other.
Note that an embarrassingly parallel problem might still be unsuited for SIMD/SIMT, since it says nothing about conditionals.

Variables: Private or public?

- Make a variable private or shared?
- Rule of thumb: Static data used by many threads shared, otherwise private
  - Note: Static data used by single threads should be private for memory management, but (usually) not at risk of corruption
- Exception: Write-access strictly necessary
  - Typically reserved for end stages of a program, e.g. summing results
- How can we handle data integrity for shared variables?



Atomics, locks, accessibility

We have several options for data integrity in concurrent programs:

- `mutex`, mutually exclusive access
  - Puts *locks* on shared variables, only allowing one thread access at a time
  - In-software solution: Data integrity is ensured at the system level
  - Usually overkill — other options are typically faster
- `atomic` operations, indivisible instruction sets
  - Combines problematic instructions into a single uninterruptible operation
  - I.e., read-modify-write cannot be interrupted by other threads
  - (If implemented) Hardware solution: Scheduler takes care of indivisibility
- Leaving sequential operations to host
  - The host will typically be a lot better at sequential code
  - Unless latter stages of program depend on modified shared value: Usually better to factorise the sequential part of the code

Any questions?

CERN