Porting codes to GPUs

Anthony Scemama

20/11/2025

Laboratoire de Chimie et Physique Quantiques

Definitions

Bandwidth and latency

Bandwidth Amount of data which goes through one point per unit of time (GB/seconds)

Latency Time to transfer *one* element between two points (seconds)

	Latency	Bandwidth
	Low $(300\ \text{km/h})^{-1}$	Low 1
	High $(80\ \text{km/h})^{-1}$	High 68

Increasing the bandwidth

https://atlanta.curbed.com/2019/4/17/18411608/ga-highway-400-express-lanes-traffic-peach-pass

Reducing latency

- Reducing latency is difficult

Reducing latency

- Reducing latency is difficult
- Hiding latency is easier:
  - Levels of caches: Smaller memories with low latencies

Reducing latency

- Reducing latency is difficult
- Hiding latency is easier:
  - Levels of caches: Smaller memories with low latencies
  - Asynchronous mechanisms
    - Prefetching data from memory to caches: The bus brings 68 people close to a motorbike

Reducing latency

- Reducing latency is difficult
- Hiding latency is easier:
  - Levels of caches: Smaller memories with low latencies
  - Asynchronous mechanisms
    - Prefetching data from memory to caches: The bus brings 68 people close to a motorbike
    - Pipelining : Instead of 1 bus going back and forth, send many busses one after the other

Reducing latency

- Reducing latency is difficult
- Hiding latency is easier:
  - Levels of caches: Smaller memories with low latencies
  - Asynchronous mechanisms
    - Prefetching data from memory to caches: The bus brings 68 people close to a motorbike
    - Pipelining : Instead of 1 bus going back and forth, send many busses one after the other
    - Out-of-order execution on CPUs

Reducing latency

- Reducing latency is difficult
- Hiding latency is easier:
  - Levels of caches: Smaller memories with low latencies
  - Asynchronous mechanisms
    - Prefetching data from memory to caches: The bus brings 68 people close to a motorbike
    - Pipelining : Instead of 1 bus going back and forth, send many busses one after the other
    - Out-of-order execution on CPUs
    - Simultaneous Multithreading (SMT)

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Latency	5 cycles
Throughput	1 per cycle

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

Latency: Numbers to keep in mind

Operation	Latency (ns)	Scaled
Int ADD	0.3	1 s
FP ADD	0.9	3 s
FP MUL	1.5	5 s
Int32 DIV	6.6	22 s
FP64 DIV	5.7	19 s
L1 cache	1.2	4 s
L2 cache	3.5	12 s
L3 cache	13	43 s
RAM	79	4 min 23 s
Send 4KB with 100 Gbps Infiniband	1 040	57 min 46 s
Send 4KB over 10 Gbps ethernet	10 000	9 h 16 min
Write 4KB randomly to NVMe SSD	30 000	1 day 4 h
Transfer 1MB to/from PCIe GPU	80 000	3 days 11 h
Random Disk Access (seek+rotation)	10 000 000	1 year 21 days

Arithmetic intensity

Arithmetic intensity : $\text{AI} = N(\text{flops})/N(\text{bytes})$

Arithmetic intensity

Arithmetic intensity : $\text{AI} = N(\text{flops})/N(\text{bytes})$

FLOPS: $\text{AI} \times \text{Bandwidth} : \frac{\text{flops}}{\text{bytes}} \times \frac{\text{bytes}}{\text{second}} = \text{flops/second}$

Arithmetic intensity

Arithmetic intensity : $\text{AI} = N(\text{flops})/N(\text{bytes})$

FLOPS: $\text{AI} \times \text{Bandwidth} : \frac{\text{flops}}{\text{bytes}} \times \frac{\text{bytes}}{\text{second}} = \text{flops/second}$

- Add two vectors : $x_i = a_i + b_i$

  $\text{AI} = N/(24N) = 0.042$ flops/byte memory-bound

Arithmetic intensity

Arithmetic intensity : AI $= N(\text{flops})/N(\text{bytes})$

FLOPS: AI $\times$ Bandwidth : $\frac{\text{flops}}{\text{bytes}} \times \frac{\text{bytes}}{\text{second}} = \text{flops/second}$

- Add two vectors : $x_i = a_i + b_i$
  AI $= N/(24N) = 0.042$ flops/byte memory-bound
- Dot product : $x = \sum_i^N a_i \times b_i$
  AI $= 2N/(16N) = 0.125$ flops/byte memory-bound

Arithmetic intensity

Arithmetic intensity : AI $= N(\text{flops})/N(\text{bytes})$

FLOPS: AI $\times$ Bandwidth : $\frac{\text{flops}}{\text{bytes}} \times \frac{\text{bytes}}{\text{second}} = \text{flops/second}$

- Add two vectors : $x_i = a_i + b_i$
  AI $= N/(24N) = 0.042$ flops/byte memory-bound
- Dot product : $x = \sum_i^N a_i \times b_i$
  AI $= 2N/(16N) = 0.125$ flops/byte memory-bound
- Matrix-vector: $X_i = \sum_j^N A_{ij} \times b_j$
  AI $= 2N^2/(8N^2 + 16N) \sim 0.25$ flops/byte memory-bound

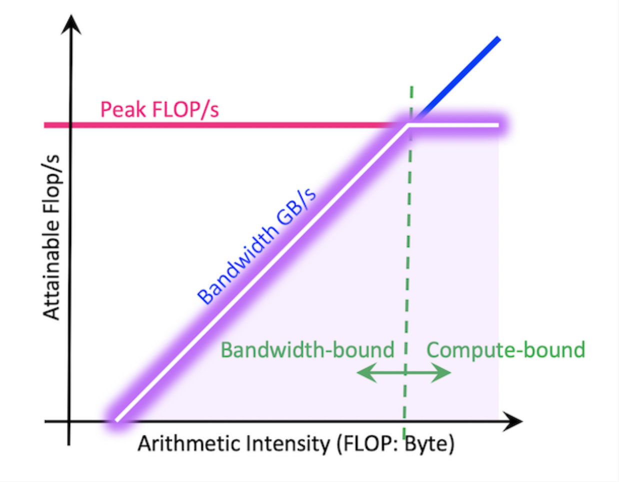
Arithmetic intensity

Arithmetic intensity : AI $= N(\text{flops})/N(\text{bytes})$

FLOPS: AI $\times$ Bandwidth : $\frac{\text{flops}}{\text{bytes}} \times \frac{\text{bytes}}{\text{second}} = \text{flops/second}$

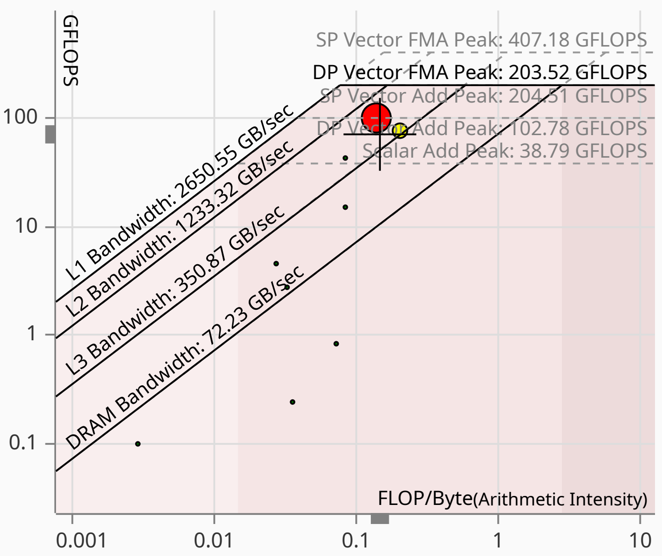
- Add two vectors : $x_i = a_i + b_i$
  AI $= N/(24N) = 0.042$ flops/byte memory-bound
- Dot product : $x = \sum_i^N a_i \times b_i$
  AI $= 2N/(16N) = 0.125$ flops/byte memory-bound
- Matrix-vector: $X_i = \sum_j^N A_{ij} \times b_j$
  AI $= 2N^2/(8N^2 + 16N) \sim 0.25$ flops/byte memory-bound
- Matrix-matrix :$X_{ij} = \sum_k A_{ik} \times B_{kj}$
  AI $= 2N^3/(24N^2) \propto N$ CPU-bound

Roofline model



https://docs.nersc.gov/tools/performance/roofline

Roofline model (CPU)



GPUs

What is a GPU?

- GPU: Graphics Processing Unit

What is a GPU?

- GPU: Graphics Processing Unit
- Accelerator designed for computer graphics

What is a GPU?

- GPU: Graphics Processing Unit
- Accelerator designed for computer graphics
- Images, Video, 3D: huge arrays of independent pixels, vertices, or voxels

	GPU	CPU
Optimized for	High-throughput parallel processing	Low-latency sequential processing
Typical workload	Repetitive numeric ops on large data arrays	Complex logic, OS, control flow
# Cores	$\mathcal{O}(10^4)$	$\mathcal{O}(10^2)$

Comparison GPU/CPU

GPU — many simple parallel tasks

CPU — few complex coordinated tasks

Comparison GPU/CPU

Processing unit	**Nvidia H100 SXM**	**AMD EPYC™ 9754**
Cores	16896	128
Clock speed	1.6 GHz	2.25 GHz
Release date	September 2022	June 2023
FP64	34 TFlop/s	6.9 TFlop/s
FP64 (tensor core)	67 TFlop/s	–
FP32	67 TFlop/s	13.8 TFlop/s
TF32 (tensor core)	989 TFlop/s	–
FP16 (tensor core)	1979 TFlop/s	–
FP8 (tensor core)	3958 TFlop/s	–
Memory	80 GB	max 6000 GB
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)
Interconnection Nvlink	900 GB/s	
Interconnection PCIe	128 GB/s	
Quantum X800 Infiniband network	100 GB/s	100 GB/s
TDP	700 W	360 W

Comparison GPU/CPU

Processing unit	**Nvidia H100 SXM**	**AMD EPYC™ 9754**	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	
FP64 (tensor core)	67 TFlop/s	–	
FP32	67 TFlop/s	13.8 TFlop/s	
TF32 (tensor core)	989 TFlop/s	–	
FP16 (tensor core)	1979 TFlop/s	–	
FP8 (tensor core)	3958 TFlop/s	–	
Memory	80 GB	max 6000 GB	
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	Nvidia H100 SXM	AMD EPYC™ 9754	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	
FP16 (tensor core)	1979 TFlop/s	–	
FP8 (tensor core)	3958 TFlop/s	–	
Memory	80 GB	max 6000 GB	
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s	128 GB/s	
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	**Nvidia H100 SXM**	**AMD EPYC™ 9754**	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	
FP16 (tensor core)	1979 TFlop/s	–	
FP8 (tensor core)	3958 TFlop/s	–	
Memory	80 GB	max 6000 GB	
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	Nvidia H100 SXM	AMD EPYC™ 9754	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	$\leftarrow \times 71$
FP16 (tensor core)	1979 TFlop/s	–	
FP8 (tensor core)	3958 TFlop/s	–	
Memory	80 GB	max 6000 GB	
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s	128 GB/s	
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	**Nvidia H100 SXM**	**AMD EPYC™ 9754**	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	$\leftarrow \times 71$
FP16 (tensor core)	1979 TFlop/s	–	$\leftarrow \times 143$
FP8 (tensor core)	3958 TFlop/s	–	$\leftarrow \times 286$
Memory	80 GB	max 6000 GB	
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	**Nvidia H100 SXM**	**AMD EPYC™ 9754**	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	$\leftarrow \times 71$
FP16 (tensor core)	1979 TFlop/s	–	$\leftarrow \times 143$
FP8 (tensor core)	3958 TFlop/s	–	$\leftarrow \times 286$
Memory	80 GB	max 6000 GB	$\leftarrow \times 1/10 - 1/75$
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	Nvidia H100 SXM	AMD EPYC™ 9754	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	$\leftarrow \times 71$
FP16 (tensor core)	1979 TFlop/s	–	$\leftarrow \times 143$
FP8 (tensor core)	3958 TFlop/s	–	$\leftarrow \times 286$
Memory	80 GB	max 6000 GB	$\leftarrow \times 1/10 - 1/75$
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	$\leftarrow \times 7$
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	

Comparison GPU/CPU

Processing unit	Nvidia H100 SXM	AMD EPYC™ 9754	
Cores	16896	128	$\leftarrow \times 132$
Clock speed	1.6 GHz	2.25 GHz	
Release date	September 2022	June 2023	
FP64	34 TFlop/s	6.9 TFlop/s	$\leftarrow \times 5$
FP64 (tensor core)	67 TFlop/s	–	$\leftarrow \times 10$
FP32	67 TFlop/s	13.8 TFlop/s	$\leftarrow \times 5$
TF32 (tensor core)	989 TFlop/s	–	$\leftarrow \times 71$
FP16 (tensor core)	1979 TFlop/s	–	$\leftarrow \times 143$
FP8 (tensor core)	3958 TFlop/s	–	$\leftarrow \times 286$
Memory	80 GB	max 6000 GB	$\leftarrow \times 1/10 - 1/75$
Memory Bandwidth	3350 GB/s (HBM3)	460.8 GB/s (DDR5)	$\leftarrow \times 7$
Interconnection Nvlink	900 GB/s		
Interconnection PCIe	128 GB/s		
Quantum X800 Infiniband network	100 GB/s	100 GB/s	
TDP	700 W	360 W	$\leftarrow \times 2$

Double Precision ridge point of roofline model

CPU

- Add/Mul: $\frac{3.45 \text{ TFlops/s}}{0.46 \text{ TBytes/s}} = 7.5$ Flops/Byte
- FMA: $\frac{6.9 \text{ TFlops/s}}{0.46 \text{ TBytes/s}} = 15$ Flops/Byte
  DP Matrix multiplication: $240 \times 240 \times 240$

Double Precision ridge point of roofline model

CPU

- Add/Mul: $\frac{3.45 \text{ TFlops/s}}{0.46 \text{ TBytes/s}} = 7.5$ Flops/Byte
- FMA: $\frac{6.9 \text{ TFlops/s}}{0.46 \text{ TBytes/s}} = 15$ Flops/Byte
  DP Matrix multiplication: $240 \times 240 \times 240$

GPU

- DP Cuda cores: $\frac{34 \text{ TFlops/s}}{3.35 \text{ TBytes/s}} = 10$ Flops/Byte
- DP Tensor cores: $\frac{67 \text{ TFlops/s}}{3.35 \text{ TBytes/s}} = 20$ Flops/Byte
  DP Matrix multiplication: $320 \times 320 \times 320$
- SP Tensor cores: $\frac{989 \text{ TFlops/s}}{3.35 \text{ TBytes/s}} = 295$ Flops/Byte
  SP Matrix multiplication: $2362 \times 2362 \times 2362$

Tensor cores

Tensor core : Specialized processing unit to perform accumulations of small matrix multiplications ($4 \times 4 \times 4$ MMA):

$$D = A.B + C$$

- A, B, C, D: matrices

Tensor cores

Tensor core : Specialized processing unit to perform accumulations of small matrix multiplications ($4 \times 4 \times 4$ MMA):

$$D = A.B + C$$

- A, B, C, D: matrices
- Operation in low-precision, with accumulation in high-precision

Tensor cores

Tensor core : Specialized processing unit to perform accumulations of small matrix multiplications ($4 \times 4 \times 4$ MMA):

$$D = A.B + C$$

- A, B, C, D: matrices
- Operation in low-precision, with accumulation in high-precision
- Throughput: one per cycle

Tensor cores

Tensor core : Specialized processing unit to perform accumulations of small matrix multiplications ($4 \times 4 \times 4$ MMA):

$$D = A.B + C$$

- A, B, C, D: matrices
- Operation in low-precision, with accumulation in high-precision
- Throughput: one per cycle
- Multiple tensor cores can operate together to perform up to $64 \times 16 \times 64$

See: `https://semianalysis.com/2025/06/23/nvidia-tensor-core-evolution-from-volta-to-blackwell/`

Tensor cores

VOLTA MMA.SYNC

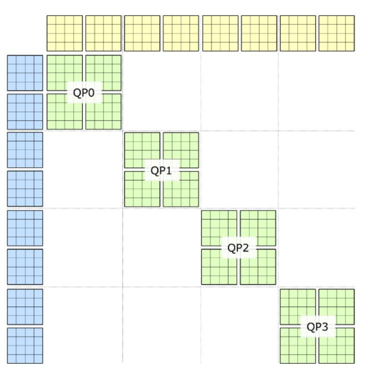
Warp-scoped matrix multiply instruction

Warp is partitioned into Quad Pairs

- **QP0:** `T0..T3` `T16..T19`
- **QP1:** `T4..T7` `T20..T23`
- **QP2:** `T8..T11` `T24..T27`
- **QP3:** `T12..T15` `T28..T31`

(eight threads each)

Each Quad Pair performs one **8-by-8-by-4** matrix multiply



65 NVIDIA

https://www.olcf.ornl.gov/wp-content/uploads/2019/11/ORNL_Tensor_Core_Training_Aug2019.pdf

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU
Register access	5.2 ns	0.3 ns
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns
L1 access	25 ns	1.2 ns
L2 access	260 ns	3.5 ns
Memory access	520 ns	79 ns
Launch a CUDA kernel or a function	10 μs	1-2 ns
CPU-GPU transfer	$\sim$ 10 μs	

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU	
Register access	5.2 ns	0.3 ns	$\leftarrow \times 17$
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns	$\leftarrow \times 5$
L1 access	25 ns	1.2 ns	
L2 access	260 ns	3.5 ns	
Memory access	520 ns	79 ns	
Launch a CUDA kernel or a function	10 μs	1-2 ns	
CPU-GPU transfer	$\sim$ 10 μs		

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU	
Register access	5.2 ns	0.3 ns	$\leftarrow \times 17$
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns	$\leftarrow \times 5$
L1 access	25 ns	1.2 ns	$\leftarrow \times 21$
L2 access	260 ns	3.5 ns	$\leftarrow \times 74$
Memory access	520 ns	79 ns	
Launch a CUDA kernel or a function	10 μs	1-2 ns	
CPU-GPU transfer	$\sim$ 10 μs		

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU	
Register access	5.2 ns	0.3 ns	$\leftarrow \times 17$
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns	$\leftarrow \times 5$
L1 access	25 ns	1.2 ns	$\leftarrow \times 21$
L2 access	260 ns	3.5 ns	$\leftarrow \times 74$
Memory access	520 ns	79 ns	$\leftarrow \times 7$
Launch a CUDA kernel or a function	10 μs	1-2 ns	
CPU-GPU transfer	$\sim$ 10 μs		

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU	
Register access	5.2 ns	0.3 ns	$\leftarrow \times 17$
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns	$\leftarrow \times 5$
L1 access	25 ns	1.2 ns	$\leftarrow \times 21$
L2 access	260 ns	3.5 ns	$\leftarrow \times 74$
Memory access	520 ns	79 ns	$\leftarrow \times 7$
Launch a CUDA kernel or a function	10 μs	1-2 ns	$\leftarrow \times 5000$
CPU-GPU transfer	$\sim$ 10 μs		

GPU/CPU Latencies

Latency is problem #1 with GPU computing.

	GPU	CPU	
Register access	5.2 ns	0.3 ns	$\leftarrow \times 17$
FMA ($c \leftarrow a \times b + c$)	5.2 ns	1 ns	$\leftarrow \times 5$
L1 access	25 ns	1.2 ns	$\leftarrow \times 21$
L2 access	260 ns	3.5 ns	$\leftarrow \times 74$
Memory access	520 ns	79 ns	$\leftarrow \times 7$
Launch a CUDA kernel or a function	10 μs	1-2 ns	$\leftarrow \times 5000$
CPU-GPU transfer	$\sim$ 10 μs		

Asynchronous methods are mandatory: Many more threads than cores

Comparison GPU/CPU

"100x acceleration" with GPU implies

Comparison GPU/CPU

"100x acceleration" with GPU implies

1. Low-precision matrix multiplication $\Longrightarrow$ OK

Comparison GPU/CPU

"100x acceleration" with GPU implies

1. Low-precision matrix multiplication $\Longrightarrow$ OK
2. Unfair comparison with the CPU $\Longrightarrow$ not OK
    - Host CPU, not designed for HPC
    - Comparison to single-core CPU code

Comparison GPU/CPU

"100x acceleration" with GPU implies

1. Low-precision matrix multiplication $\Longrightarrow$ OK
2. Unfair comparison with the CPU $\Longrightarrow$ not OK
   - Host CPU, not designed for HPC
   - Comparison to single-core CPU code
3. Poor efficiency of the reference CPU code $\Longrightarrow$ not OK

Comparison GPU/CPU

"100x acceleration" with GPU implies

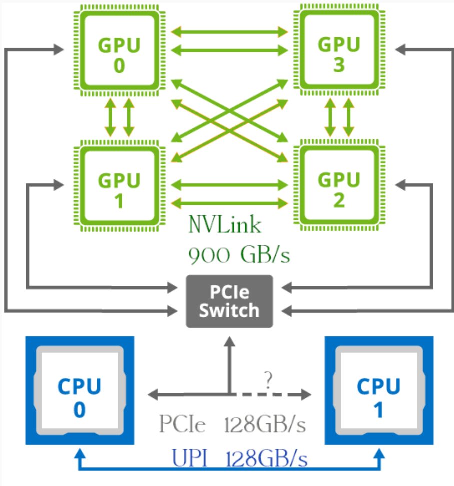
1. Low-precision matrix multiplication $\implies$ OK
2. Unfair comparison with the CPU $\implies$ not OK
   - Host CPU, not designed for HPC
   - Comparison to single-core CPU code
3. Poor efficiency of the reference CPU code $\implies$ not OK

Acceptable range

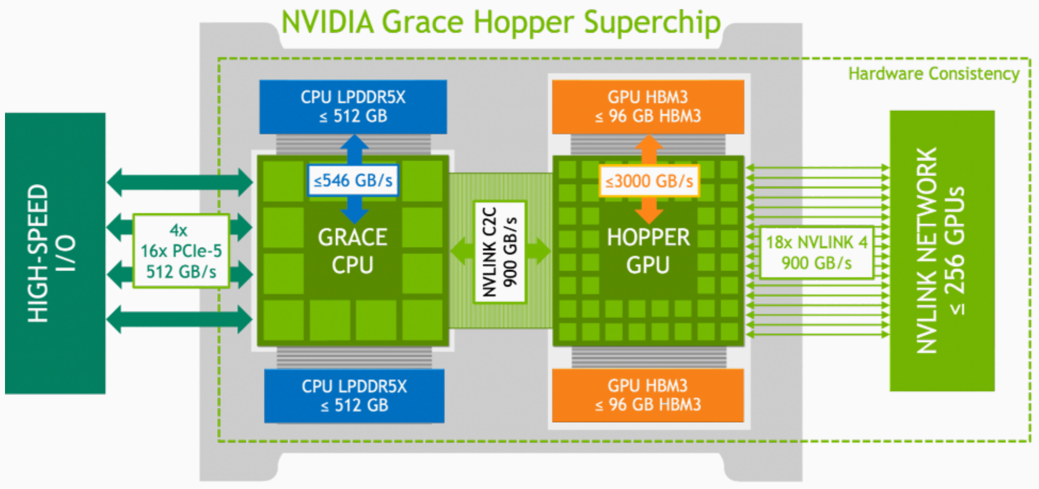
- Between $0.1\times$ and $10\times$
- ≥ 4 is very good with double precision

Programming GPUs

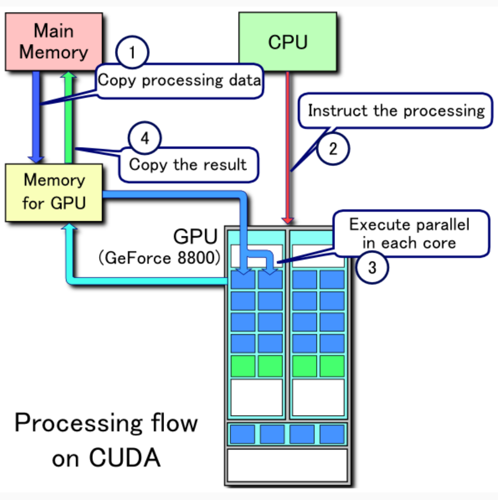
Architecture of a typical GPU node



Architecture of a Grace Hopper Superchip



Processing flow



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

Latency: Numbers to keep in mind

Operation	Latency (ns)	Scaled
Int ADD	0.3	1 s
FP ADD	0.9	3 s
FP MUL	1.5	5 s
Int32 DIV	6.6	22 s
FP64 DIV	5.7	19 s
L1 cache	1.2	4 s
L2 cache	3.5	12 s
L3 cache	13	43 s
RAM	79	4 min 23 s
Send 4KB with 100 Gbps Infiniband	1 040	57 min 46 s
Send 4KB over 10 Gbps ethernet	10 000	9 h 16 min
Write 4KB randomly to NVMe SSD	30 000	1 day 4 h
Transfer 1MB to/from PCIe GPU	80 000	3 days 11 h
Random Disk Access (seek+rotation)	10 000 000	1 year 21 days

Example

Multiplication of $10\ 000 \times 10\ 000$ matrices

Example

Multiplication of $10\ 000 \times 10\ 000$ **matrices**

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 10\ 000^2 \times 8}{128\,10^9} = 12.5$ ms

Example

Multiplication of $10\ 000 \times 10\ 000$ **matrices**

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 10\ 000^2 \times 8}{128\,10^9} = 12.5$ ms
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 10\ 000^3}{67\,10^{12}} \sim 30$ ms

Example

Multiplication of $10\ 000 \times 10\ 000$ **matrices**

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 10\ 000^2 \times 8}{128\,10^9} = 12.5$ ms
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 10\ 000^3}{67\,10^{12}} \sim 30$ ms
3. Bring result back on the host memory : $10^{-5} + \frac{8 \times 10\ 000^2}{128\,10^9} \sim 6.5$ ms

Example

Multiplication of $10\ 000 \times 10\ 000$ **matrices**

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 10\ 000^2 \times 8}{128\, 10^9} = 12.5$ ms
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 10\ 000^3}{67\, 10^{12}} \sim 30$ ms
3. Bring result back on the host memory : $10^{-5} + \frac{8 \times 10\ 000^2}{128\, 10^9} \sim 6.5$ ms

Total: $\sim$ 50 ms, 40% data transfer, 60% computation.

Example

Multiplication of $10\ 000 \times 10\ 000$ **matrices**

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 10\ 000^2 \times 8}{128\,10^9} = 12.5$ ms
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 10\ 000^3}{67\,10^{12}} \sim 30$ ms
3. Bring result back on the host memory : $10^{-5} + \frac{8 \times 10\ 000^2}{128\,10^9} \sim 6.5$ ms

Total: $\sim$ 50 ms, 40% data transfer, 60% computation.

On CPU: $\frac{2 \times 10\ 000^3}{6.9\,10^{12}} \sim 290$ ms : $6\times$ longer

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 8 \times 1\ 000^2}{128\, 10^9} = 135\ \mu$s

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 8 \times 1\ 000^2}{128\, 10^9} = 135\ \mu$s
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 1\ 000^3}{67\, 10^{12}} \times \frac{1}{0.5} \sim 70\ \mu$s

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 8 \times 1\ 000^2}{128\, 10^9} = 135\ \mu$s
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 1\ 000^3}{67\, 10^{12}} \times \frac{1}{0.5} \sim 70\ \mu$s
3. Bring result back on the host memory : $10^{-5} + \frac{8 \times 1\ 000^2}{128\, 10^9} = 73\ \mu$s

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

1. Transfer from host to device memory : $10^{-5} + \frac{2 \times 8 \times 1\ 000^2}{128\,10^9} = 135\ \mu$s
2. Run DGEMM kernel : $10^{-5} + \frac{2 \times 1\ 000^3}{67\,10^{12}} \times \frac{1}{0.5} \sim 70\ \mu$s
3. Bring result back on the host memory : $10^{-5} + \frac{8 \times 1\ 000^2}{128\,10^9} = 73\ \mu$s

   Total: 278 μs, 75% data transfer, 25% computation.

Small Example

Multiplication of $1\ 000 \times 1\ 000$ matrices

1. Transfer from host to device memory : $10^{-5} + \frac{2\times 8\times 1\ 000^2}{128\,10^9} = 135\ \mu$s
2. Run DGEMM kernel : $10^{-5} + \frac{2\times 1\ 000^3}{67\,10^{12}} \times \frac{1}{0.5} \sim 70\ \mu$s
3. Bring result back on the host memory : $10^{-5} + \frac{8\times 1\ 000^2}{128\,10^9} = 73\ \mu$s

   Total: 278 μs, 75% data transfer, 25% computation.

On CPU: $\frac{2\times 1\ 000^3}{6.9\,10^{12}} \sim 289\ \mu$s : no gain

Obtaining performance

1. Pack computation into very large massively parallel workloads

Obtaining performance

1. Pack computation into very large massively parallel workloads
2. Use mostly matrix multiplication to leverage tensor cores

Obtaining performance

1. Pack computation into very large massively parallel workloads
2. Use mostly matrix multiplication to leverage tensor cores
3. Use single precision or less

Obtaining performance

1. Pack computation into very large massively parallel workloads
2. Use mostly matrix multiplication to leverage tensor cores
3. Use single precision or less
4. Avoid transfering data back and forth between CPU and GPU

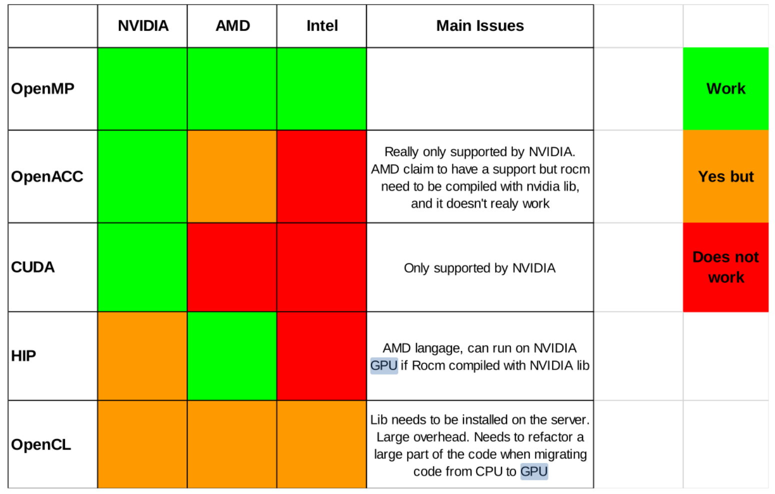
Obtaining performance

1. Pack computation into very large massively parallel workloads
2. Use mostly matrix multiplication to leverage tensor cores
3. Use single precision or less
4. Avoid transfering data back and forth between CPU and GPU
5. Make computation directly on the GPU (even if slower) when it can avoid transfers

Obtaining performance

1. Pack computation into very large massively parallel workloads
2. Use mostly matrix multiplication to leverage tensor cores
3. Use single precision or less
4. Avoid transfering data back and forth between CPU and GPU
5. Make computation directly on the GPU (even if slower) when it can avoid transfers
6. Overlap data transfer with computation

Programming models



	NVIDIA	AMD	Intel	Main Issues		
OpenMP						**Work**
OpenACC				Really only supported by NVIDIA. AMD claim to have a support but rocm need to be compiled with nvidia lib, and it doesn't realy work		**Yes but**
CUDA				Only supported by NVIDIA		**Does not work**
HIP				AMD langage, can run on NVIDIA GPU if Rocm compiled with NVIDIA lib		
OpenCL				Lib needs to be installed on the server. Large overhead. Needs to refactor a large part of the code when migrating code from CPU to GPU		

Figure 6: Language offering on GPU

Programming models

Tool	C++			Fortran		
	Intel	**AMD**	**Nvidia**	**Intel**	**AMD**	**Nvidia**
OpenACC	● ○ ○ ○ ○	● ● ● ○ ○	● ● ● ● ●	● ○ ○ ○ ○	● ● ○ ○ ○	● ● ● ● ●
OpenMP	● ● ● ● ●	● ● ● ● ●	● ● ● ● ○	● ● ● ● ●	● ● ● ● ●	● ● ● ● ●
Standard	● ● ● ● ●	● ○ ○ ○ ○	● ● ● ● ●	● ● ● ● ●	○ ○ ○ ○ ○	● ● ● ● ●
CUDA	● ● ● ● ○	● ● ● ● ○	● ● ● ● ●	○ ○ ○ ○ ○	● ○ ○ ○ ○	● ● ● ● ●
HIP	● ● ● ○ ○	● ● ● ● ●	● ● ● ● ○	○ ○ ○ ○ ○	● ○ ○ ○ ○	● ○ ○ ○ ○
SYCL	● ● ● ● ●	● ● ● ○ ○	● ● ● ○ ○	○ ○ ○ ○ ○	○ ○ ○ ○ ○	○ ○ ○ ○ ○
Alpaka	● ● ● ○ ○	● ● ● ○ ○	● ● ● ○ ○	○ ○ ○ ○ ○	○ ○ ○ ○ ○	○ ○ ○ ○ ○
Kokkos	● ● ● ○ ○	● ● ● ○ ○	● ● ● ○ ○	● ○ ○ ○ ○	● ○ ○ ○ ○	● ○ ○ ○ ○

Table 2: Vendor support compatibility on GPUs. ● ● ● ● ● : Full support with complete implementation, documentation, updates, and error handling; ● ● ● ● ○ : Indirect but comprehensive support via translation to a native model; ● ● ● ○ ○ : Partial vendor support; most features work, but some may be unavailable; ● ● ○ ○ ○ : Comprehensive support exists but is community-driven, not vendor-provided; ● ○ ○ ○ ○ : Limited support requiring significant user effort; ○ ○ ○ ○ ○ : No direct support; workarounds like custom headers or manual linking may be needed; [a]Denotes built-in parallelization by compilers.

Cuda

AXPY Offloading To a GPU using CUDA

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void axpy(REAL *x, REAL *y, int n, REAL a) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) y[id] += a * x[id];
}

int main( int argc, char* argv[] ) {

    // ... init host a, x and y
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    // Copy host vectors to device
    cudaMemcpy( d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_y, h_y, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    axpy<<<gridSize, blockSize>>>(d_x, d_y, n, a);

    // Copy array back to host
    cudaMemcpy( h_y, d_y, size, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_x);
    cudaFree(d_y);
}
```

Memory allocation on device

Memcpy from host to device

Launch parallel execution

Memcpy from device to host

Deallocation of dev memory

6

https://www.slideshare.net/slideshow/lecturegpuarchcuda04openmphomppdf/261102945

OpenMP offload

AXPY Example with OpenMP: single device

- $y = \alpha \cdot x + y$
  - x and y are vectors of size n
  - α is scalar

```
1 void axpy_ompacc(REAL* x, REAL* y, int n, REAL a) {
2   #pragma omp target device (0) map(tofrom: y[0:n]) \
3         map(to: x[0:n],a,n)
4   #pragma omp parallel for shared(x, y, n, a)
5   for (int i = 0; i < n; ++i)
6     y[i] += a * x[i];
7 }
```

- **target** directive: annotate an offloading code region
- **map** clause: map data between host and device → moving data
  - **to**|**tofrom**|**from**: mapping directions
  - Use array region

7

https://www.slideshare.net/slideshow/lecturegpuarchcuda04openmphomppdf/261102945

OpenACC

“Parallel for” on a GPU via pragmas

Option 1: OpenMP 4.5

```
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
  total = 0
#pragma omp parallel for
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

Option 2: OpenACC

```
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)
#pragma acc loop gang vector
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

https://github.com/ENCCS/OpenACC-CUDA-beginners

Kokkos

```
View<double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce("Label",
  RangePolicy< Space>(0, size),
  KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
    valueToUpdate += data(index);
  },
  sum);
```

https://github.com/kokkos/kokkos-tutorials/blob/main/Intro-Full/Slides/KokkosTutorial_ORNL20.pdf

Tensor cores

- Programming models (OpenACC, OpenMP, ...) do not generate GPU code using tensor cores

Tensor cores

- Programming models (OpenACC, OpenMP, ...) do not generate GPU code using tensor cores
- On NVidia, tensor cores require Cuda

Tensor cores

- Programming models (OpenACC, OpenMP, ...) do not generate GPU code using tensor cores
- On NVidia, tensor cores require Cuda
- On AMD, "Matrix cores" require ROCm

Tensor cores

- Programming models (OpenACC, OpenMP, ...) do not generate GPU code using tensor cores
- On NVidia, tensor cores require Cuda
- On AMD, "Matrix cores" require ROCm
- Optimal solution: Use libraries (cuBLAS, rocBLAS)

Consequences on software development

- Many different programming environments for GPUs
- Code difficult to understand for an average programmer
- Good support for C++, not for Fortran
- Performance not necessarily consistent across architectures
- Many external dependencies $\Longrightarrow$ difficult to compile/install the code

Codes

Production code

- The quantum chemist uses the executable as a tool to run simulations
- The researcher never looks at the source code
- The code can be hard to read, as long as it is efficient

Codes

Production code

- The quantum chemist uses the executable as a tool to run simulations
- The researcher never looks at the source code
- The code can be hard to read, as long as it is efficient

Research code

- The quantum chemist uses the source files as a tool to develop new methods
- They frequently look at the source code and modify it
- The code has to be easy to read
- The code needs some efficiency to test new ideas on realistic problems

Codes

Production code

- The quantum chemist uses the executable as a tool to run simulations
- The researcher never looks at the source code
- The code can be hard to read, as long as it is efficient

Research code

- The quantum chemist uses the source files as a tool to develop new methods
- They frequently look at the source code and modify it
- The code has to be easy to read
- The code needs some efficiency to test new ideas on realistic problems

Question

How to integrate GPU code in a *research* code?

My experience with GPUs

QMC: Porting the QMCkl library to GPUs

Features of QMC algorithms

- Massively parallel

Features of QMC algorithms

- Massively parallel
- Small amounts of data $\left(\mathcal{O}(N_{\text{elec}}^2), \mathcal{O}(N_{\text{elec}} \times N_{\text{MO}})\right)$

Features of QMC algorithms

- Massively parallel
- Small amounts of data $\big(\mathcal{O}(N_{\text{elec}}^2), \mathcal{O}(N_{\text{elec}} \times N_{\text{MO}})\big)$
- Low arithmetic intensity (typically < 1 Flop/byte)

Features of QMC algorithms

- Massively parallel
- Small amounts of data $\left(\mathcal{O}(N_{\text{elec}}^2), \mathcal{O}(N_{\text{elec}} \times N_{\text{MO}})\right)$
- Low arithmetic intensity (typically < 1 Flop/byte)
- One Monte Carlo step: $\leq$ 30 milliseconds
- Highly sensitive to latency

Features of QMC algorithms

- Massively parallel
- Small amounts of data $\left(\mathcal{O}(N_{\mathsf{elec}}^2), \mathcal{O}(N_{\mathsf{elec}} \times N_{\mathsf{MO}})\right)$
- Low arithmetic intensity (typically < 1 Flop/byte)
- One Monte Carlo step: $\leq$ 30 milliseconds
- Highly sensitive to latency

QMC: Porting the QMCkl library to GPUs

Experiment

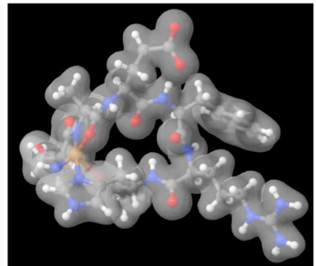


FIG. 8: Electron density of $[C_{34}H_{48}CuN_{11}O_{10}]^{2+}$.

- Reference: highly optimized CPU library

Experiment

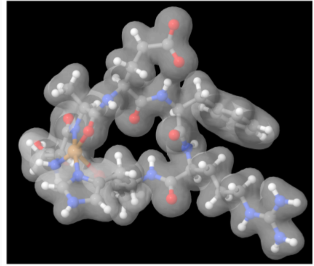


FIG. 8: Electron density of $[C_{34}H_{48}CuN_{11}O_{10}]^{2+}$.

- Reference: highly optimized CPU library
- Choose OpenMP offload for portability of the GPU code

QMC: Porting the QMCkl library to GPUs

Experiment

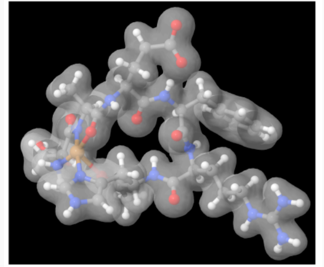


FIG. 8: Electron density of $[C_{34}H_{48}CuN_{11}O_{10}]^{2+}$.

- Reference: highly optimized CPU library
- Choose OpenMP offload for portability of the GPU code
- Benchmark: 436 electrons, 1114 AOs, 1053 MOs
- 3 kernels:
  - Computation of AOs
  - Computation of MOs
  - Computation of Jastrow factor (with eeN terms)

QMC: Porting the QMCkl library to GPUs

Results in milliseconds for 100 walkers

		AOs	MOs	Jastrow
GPU	Nvidia A100	294	1356	4650
GPU	AMD MI250	399	2409	5314
CPU	Intel Skylake 52 cores	31	204	1124

QMC: Porting the QMCkl library to GPUs

Results in milliseconds for 100 walkers

		AOs	MOs	Jastrow
GPU	Nvidia A100	294	1356	4650
GPU	AMD MI250	399	2409	5314
CPU	Intel Skylake 52 cores	31	204	1124

Total failure: runs slower on GPU than CPU!

Why a failure?

- Too small workloads $\Longrightarrow$ high sensitivity to latency

QMC: Porting the QMCkl library to GPUs

Why a failure?

- Too small workloads $\Longrightarrow$ high sensitivity to latency
- Low arithmetic intensity:

Why a failure?

- Too small workloads $\Longrightarrow$ high sensitivity to latency
- Low arithmetic intensity:
  - AOs: linear scaling algorithm, complex logic, few flops
  - MOs: Sparse-dense matrix multiplication $(436 \times 1114 \times 1053)$

Why a failure?

- Too small workloads $\Longrightarrow$ high sensitivity to latency
- Low arithmetic intensity:
    - AOs: linear scaling algorithm, complex logic, few flops
    - MOs: Sparse-dense matrix multiplication $(436 \times 1114 \times 1053)$
- No use of tensor cores

Still work in progress. . .

Coupled Cluster in Quantum Package

Features of CCSD

- High arithmetic intensity: $\mathcal{O}(N^6)$ compute, $\mathcal{O}(N^4)$ data

Coupled Cluster in Quantum Package

Features of CCSD

- High arithmetic intensity: $\mathcal{O}(N^6)$ compute, $\mathcal{O}(N^4)$ data
- Bottleneck: tensor contractions
- Tensor contractions can be recast into transpositions + matrix multiplications:

$$\Omega_{aibj} = \bar{g}_{aibj} + \sum_{cd} t_{ij}^{cd}\, \bar{g}_{acbd}$$

$$O_{ab,ij} = G_{ab,ij} + \sum_{cd} G_{ab,cd}.T^{\dagger}_{cd,ij}$$

Coupled Cluster in Quantum Package

Features of CCSD

- High arithmetic intensity: $\mathcal{O}(N^6)$ compute, $\mathcal{O}(N^4)$ data
- Bottleneck: tensor contractions
- Tensor contractions can be recast into transpositions + matrix multiplications:

$$\Omega_{aibj} = \bar{g}_{aibj} + \sum_{cd} t_{ij}^{cd} \, \bar{g}_{acbd}$$

$$O_{ab,ij} = G_{ab,ij} + \sum_{cd} G_{ab,cd} . T^{\dagger}_{cd,ij}$$

- Transposition: $\mathcal{O}(N^4)$ compute + data $\Longrightarrow$ High bandwidth
- Matrix multiplications: $\mathcal{O}(N^6)$ compute, $\mathcal{O}(N^4)$ data $\Longrightarrow$ Tensor cores

Porting the CCSD code to GPU

Lessons learned from QMCkl

- Avoid OpenMP offload:
  - It does not take advantage of tensor cores
  - It requires a GPU-compatible compiler.
  - NVFortran could not compile QP because it does not understand some intrinsic functions (`trailz`, `shiftr`, `shiftl`)
- Programming model: use only cuBLAS or rocBLAS
- Advantages: Independent of the compiler, only a few libraries to add at link stage

Porting the CCSD code to GPU

1. Creation of a little C GPU library, and ISO-C-Bindings that calls cuBLAS

Porting the CCSD code to GPU

1. Creation of a little C GPU library, and ISO-C-Bindings that calls cuBLAS to
    - Allocate/deallocate arrays on the GPU (unified memory)
    - Copy data to/from host to device
    - Select which GPU runs the following commands
    - Perform asynchronous matrix operations (GEMM, GEMV, DOT) in double or single precision
    - Perform matrix transpositions using `cublas_dgeam`
    - Task synchronization
2. Create a CPU library with the exact same API, but working only on the CPU

Porting the CCSD code to GPU

1. Creation of a little C GPU library, and ISO-C-Bindings that calls cuBLAS to
    - Allocate/deallocate arrays on the GPU (unified memory)
    - Copy data to/from host to device
    - Select which GPU runs the following commands
    - Perform asynchronous matrix operations (GEMM, GEMV, DOT) in double or single precision
    - Perform matrix transpositions using `cublas_dgeam`
    - Task synchronization
2. Create a CPU library with the exact same API, but working only on the CPU
3. Change the CCSD code to use the library

Benzene cc-pVTZ

Name	#cores	#GPUs	Time (s)	Peak (TF)
Intel Xeon Gold 6130 CPU @ 2.10GHz	32	0	693	2.15
AMD EPYC 7402	48	0	695	2.15
Intel Xeon Gold 6140 CPU @ 2.30GHz	36	0	703	2.65
+ V100 GPU		1	208 (x3.4)	7.8 (x2.9)
ARM Neoverse v2 + A100 GPU	80	0	767	1.92
	80	1	112 (x7)	19.5 (x10)

- Intel Xeon Gold + V100: CALMIP
- ARM Neoverse + A100: Turpan @ MesoNet

Benzene cc-pVQZ

Name	#cores	#GPUs	Time (s)	Peak (TF)
Intel Xeon Gold 6140 CPU @ 2.30GHz	36	0	11062	2.65
+ V100 GPU		1	38064 (x0.3)	7.8 (x3)
ARM Neoverse v2 + A100 GPU	80	0	13189	1.92
	80	1	1635 (x8)	19.5 (x10)

- Slower on V100: only 16GB on V100, 80GB on A100

Success!!

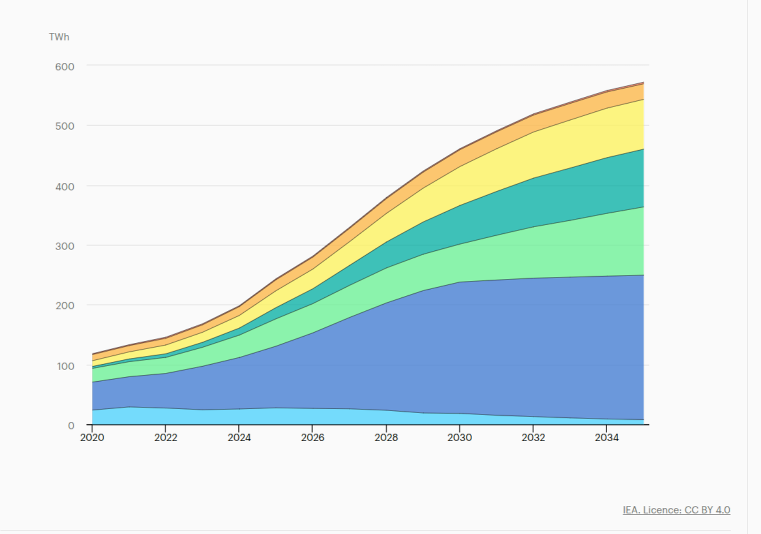
Green?

Are GPUs Green?

- Less energy-intensive than CPUs for compute-instensive workflows (High-Performance Linpack, Matrix multiplications)
- For low-energy jobs (High-Performance Conjugate-Gradients): not much better and often worse

	Fugaku	Leonardo	Aurora	El Capitan
Year	2020	2023	2023	2024
Architecture	CPU (ARM)	GPU (Nvidia)	GPU (Intel)	GPU (AMD)
Watts	29.9 MW	7.4 MW	38.7 MW	29.6 MW
HPL	442 PFlops	241 PFlops	1012 PFlops	1742 PFlops
	14.8 GFlops/W	32.6 GFlops/W	26.1 GFlops/W	58.9 GFlops/W
HPCG	16 PFlops	3.1 PFlops	5.6 PFlops	17 PFlops
	535 MFlops/W	419 MFlops/W	145 MFlops/W	574 MFlops/W

Are GPUs Green?



IEA (2025), Electricity generation for data centres by fuel in the United States, Base Case, 2020-2035, IEA, Paris `https://www.iea.org/data-and-statistics/charts/electricity-generation-for-data-centres-by-fuel-in-the-united-states-base-case-2020-2035`, Licence: CC BY 4.0