Data Processing on Modern Hardware

Jana Giceva

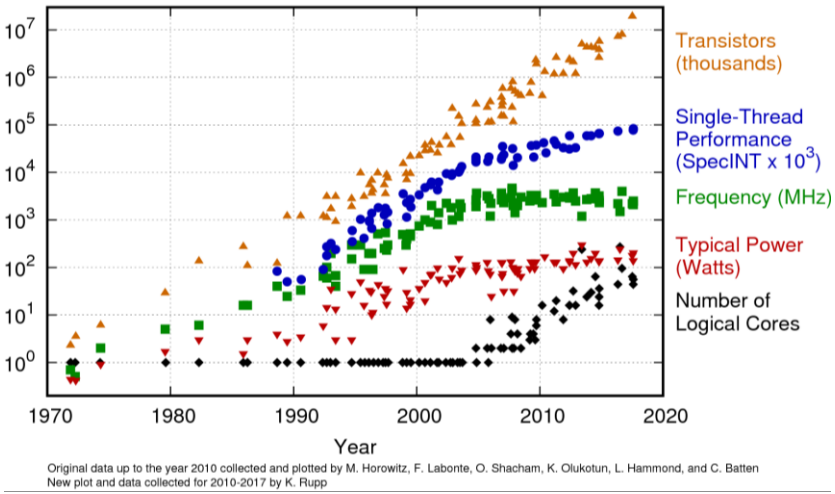
Lecture 8: GPUs

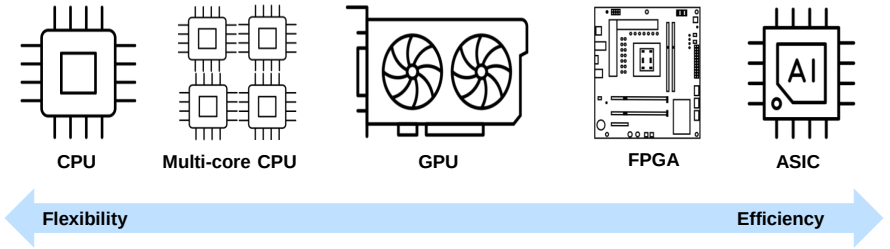
Intro to GPU computing

Moore’s Law (recap)

- Increase of transistors / CPU does not yield single-core performance
- Performance capped due to economic feasibility, power, and safety

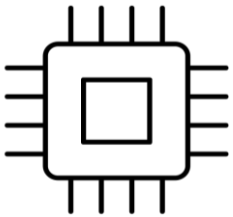
- Q: Is the solution to add more cores (multicore lecture)?
  - Only partially. Why?



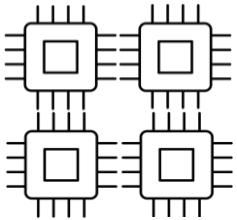


- **Specialization** – give up generality, for better efficiency and speed-up certain operations.
- Various **accelerators** on the spectrum.

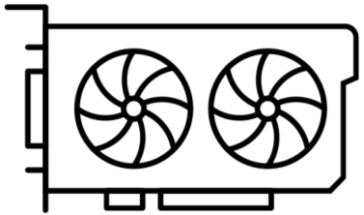
Accelerators



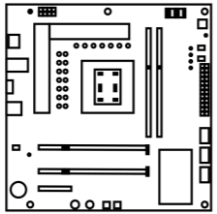
CPU



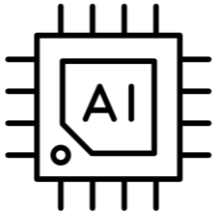
Multi-core CPU



GPU



FPGA



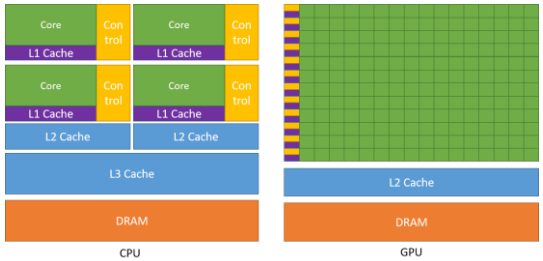
ASIC

Flexibility — **Efficiency**

- The idea of accelerators is to leverage a new architecture as an alternative.
- But, new hardware > different programming model → increased complexity.
- In GPUs case, the efficiency is reached through high level of parallelism.
  - Throughput-oriented design
  - Often aimed at scientific workloads. Why?

Graphics Processing Unit (GPU)

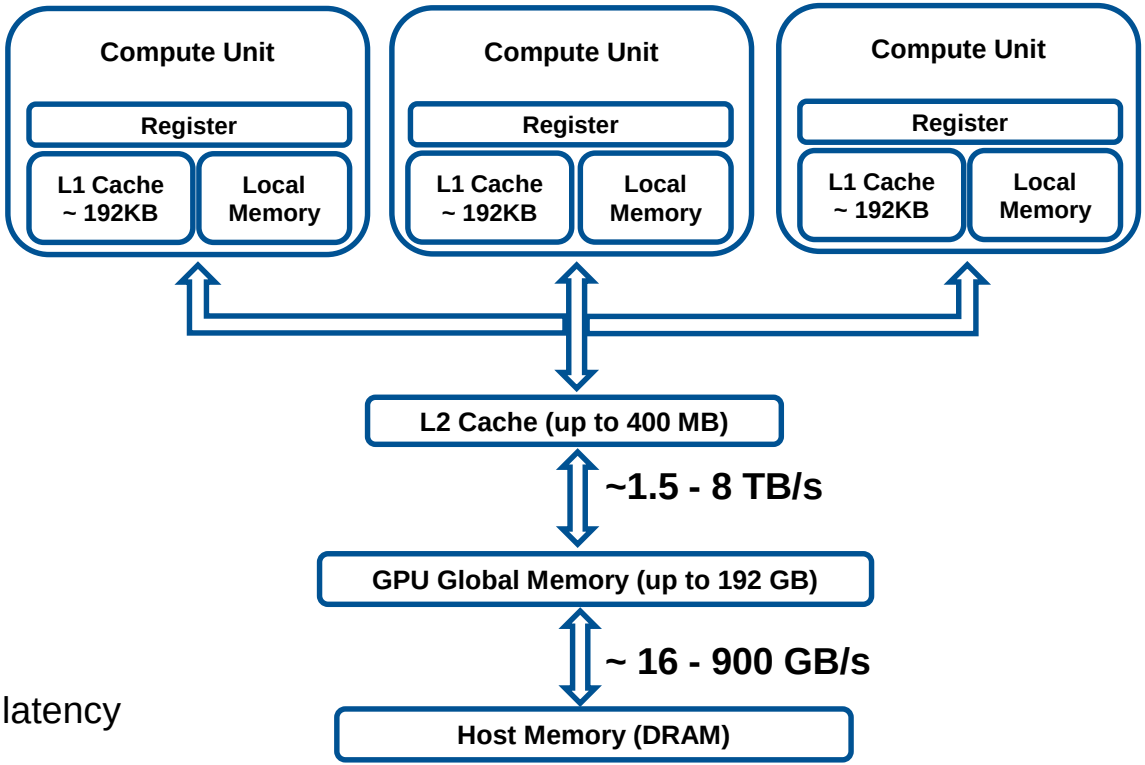
- Highly specialized co-processing chip
  - Primary target: Image rendering → parallel computations
- **Performance increased ~2.4x yearly**
  (during Moore's Law, CPUs performance increased 1.8x)
- **GPGPUs are general purpose GPUs**
  - CPU cores != GPU cores → fundamental design differences



- **CPU cores:**
  - Latency-oriented design
  - Tens of cores
  - Minimize the latency of arith ops
  - Bandwidth: ~100 GB/s
- **GPU cores:**
  - Throughput-oriented design
  - Thousands of cores
  - Large number of FLOPS
  - Bandwidth: ~ 3 TB/s (H100), ~6 TB/s (GB200)

GPU architecture

- **Compute units**
  - Computation cores
  - Register files
  - L1 cache
  - Shared memory
- **On-chip memory**
  - L1 cache, shared memory
  - L2 cache
- **On-device memory**
  - GPU global memory
  - High bandwidth, but also high latency



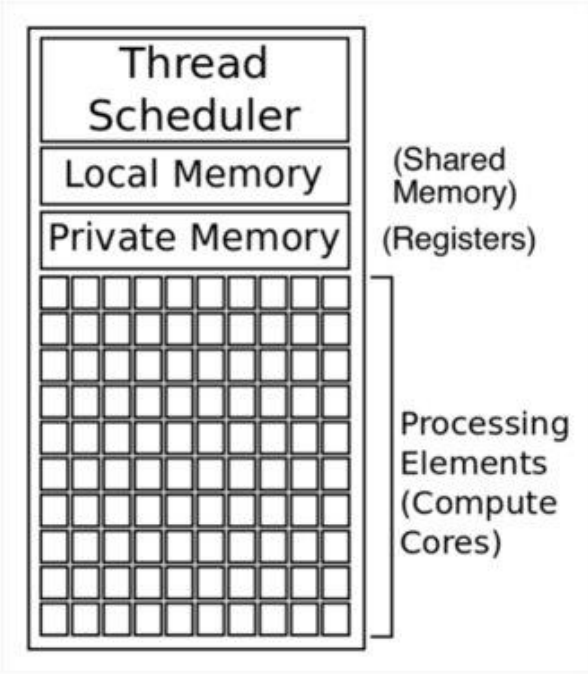
Terminology

- Different vendors have different nomenclature
- NVIDIA and AMD are the biggest GPGPU vendors as of today

Host	AMD GPU	NVIDIA GPU	Intel Gen11	OpenCL
CPU	GPU	GPU	GPU	Compute Device
Multi-processor	Compute Unit	Streaming Multiprocessor	Subslice	Compute Unit
Core	Processing Element	Compute Core	Execution Unit	Processing Element
Thread	Work Item	Thread	/	Work Item
Vector	Vector	SIMT Warp	SIMD	Vector

GPU compute unit [vendor independent]

- Compute cores
  - 32 or more
- Registers
  - Private memory per core
- Local memory
  - Shared for all cores in the compute unit
- Thread scheduler
  - Assigns threads or thread groups to cores



Parallel and High Performance Computing

Streaming Multiprocessor [NVIDIA A100]

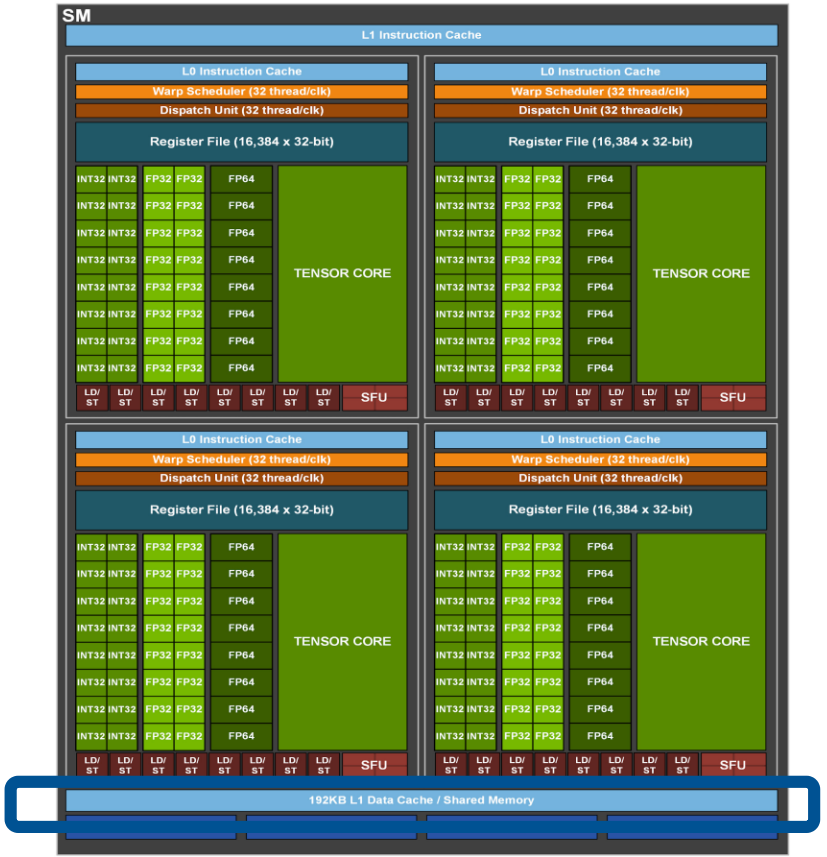
- **Computation cores**
  - CUDA or Tensor cores
  - GPU equivalent of a vector lane
- **Streaming multiprocessor (SM)**
  - GPU equivalent of a full CPU core
  - 64 CUDA cores per SM → 6’912 in A100
  - 4 Tensor cores per SM → 432 in A100
- **L1 cache + shared memory**
- **Warp scheduler**
- **Register Files**
- **Dispatch Unit**

L1 cache + Shared Memory – SM [A100]

- Combined since the Volta architecture (ca. 2018)

- Before unified L1 cache and shared memory:
  - L1 cache was managed by the OS
  - Shared memory was managed explicitly by the code

- Now:
  - No cache misses
  - Can be configured for each SM separately
  - Same address space
  - Shared memory serves as scratchpad memory
  - A100 SM: up to 164 KiB for shared memory

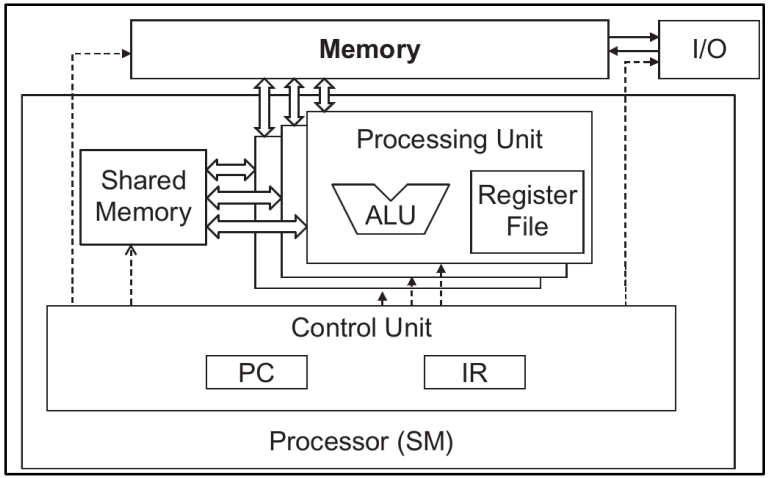


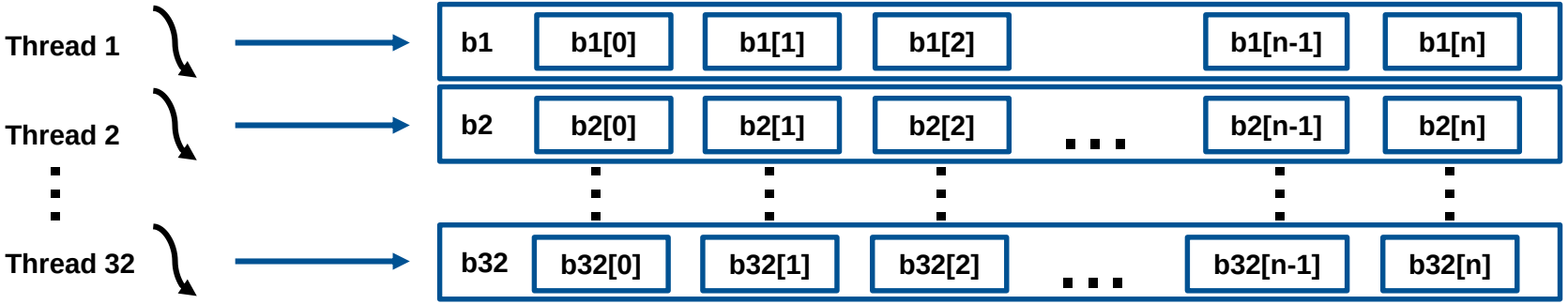
Register File [A100]

- Sends data directly to the computation core
- Does not invoke load instructions to fetch data

- Organized in 32 banks
- Each bank stores data private to a single thread

- Two orders of magnitude lower access latency compared to global memory





Warp Scheduler – [A100]

- Assigns groups of threads in an execution queue
- All threads in the group must execute the same instruction
- The warp scheduler assigns a single thread per core

- Current designs allow a whole thread group to be executed in a single cycle
  - 32 threads per group
  - A100 (server-grade GPU)
    - SM count: 108, CUDA core count: 6’912 → 64 cores/SM
    - 2 warps in a single cycle per SM → 216 per GPU

GPU Programming Model
Physical execution

Physical Execution Model

- **In the stream multiprocessor (SM), threads are scheduled in warps**
  - Each warp has 32 threads
  - One warp is executed in a single SM
- **All threads in a warp:**
  - Correspond to the same thread block
  - Execute the same instruction in parallel
- **Warps share the control unit in the SM**
- **Max active threads** = threads per warp * warps per SM * SMs
- For NVIDIA A100 = 32 * 64 * 108 = 221 184

Independent Threads Scheduling

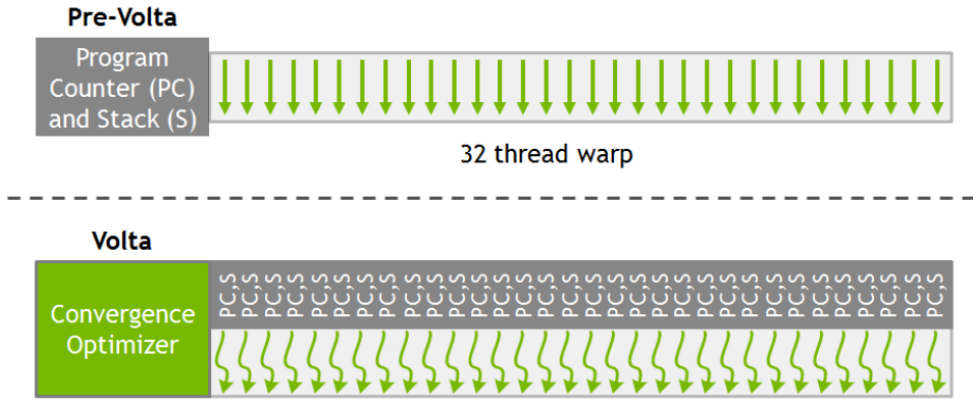
- All threads within a warp *execute the same instructions*

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

- When one thread enters a branch, all threads have to execute it
  - The effect is called *branch divergence*
  - All threads are active

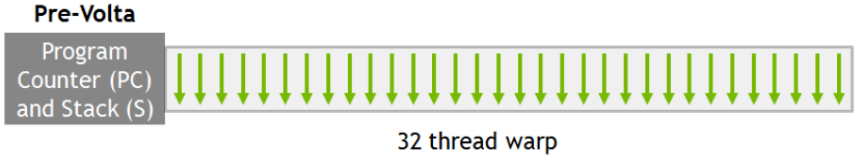
- **Lock step execution (pre-Volta)**

- **Interleaved execution model (>Volta)**



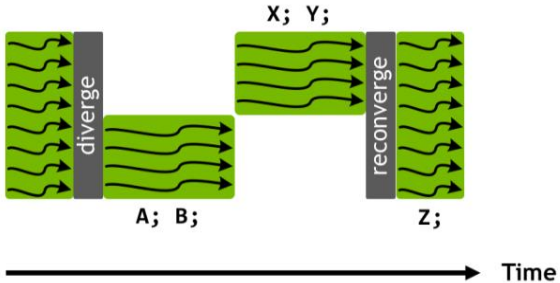
Lock-step execution model

- **Maintains execution state per warp (32 threads)**
  - Single program counter and call stack
  - Reduced resources to track thread state



- **Divergent branches are serialized**
  - All statements within a branch are executed until completion
  - A mask is used to detect active threads
  - The mask is stored until threads converge

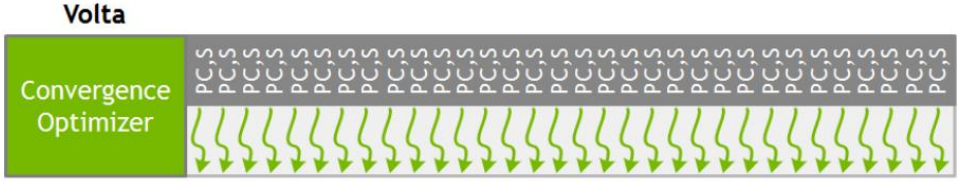
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



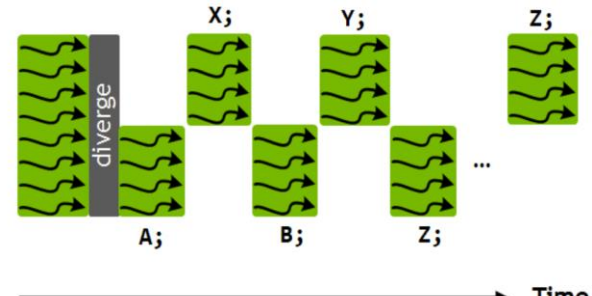
- **Thread lose concurrency**
  - Using locks/mutexes can lead to bottlenecks
  - No fine-grained parallelism

Interleaved Execution Model

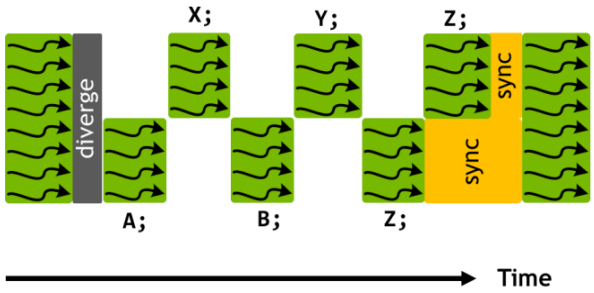
- **Maintains execution state per thread**
  - Program counter and call stack
  - Convergence optimizer handles exec. order
- **Interleaved execution across branches**
- **Thread communication within a warp**
- **Does Z reconverge automatically?**
  - Data required from another thread
  - No synchronization → auto-reconverge
- Explicit reconvergence
  - __syncwarp() ensures convergence after Z
  - **What happens if the call is made before Z?**



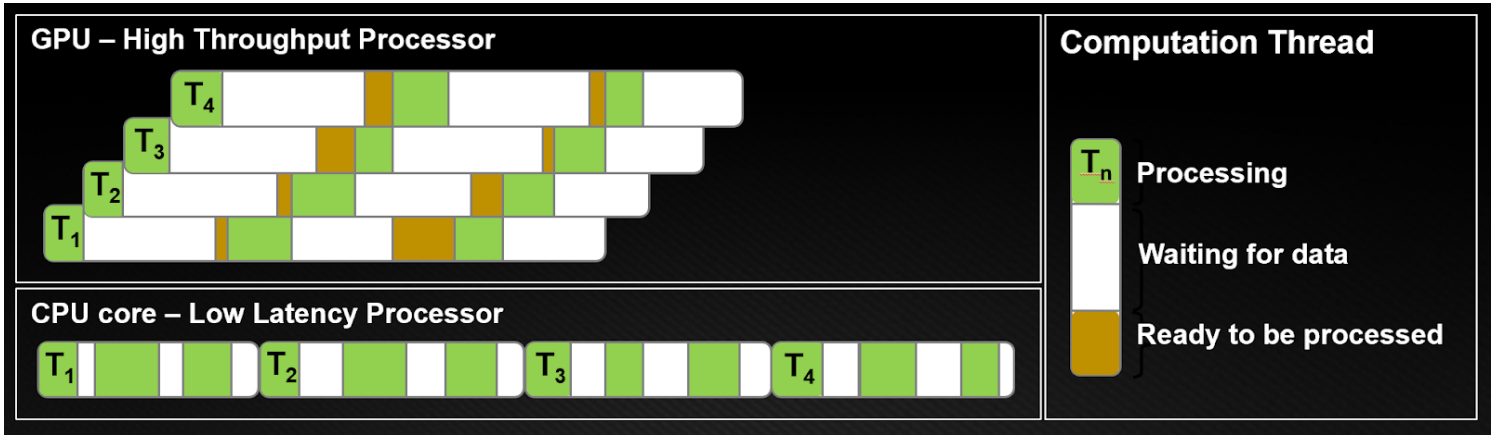

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```




```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Warp execution



- **Parallel execution of warps to mask memory access penalties**
  - L1 cache latency = 28 cycles (CPU was 4 cycles)
  - Global GPU memory latency ~ 350 cycles (CPU was 50-70 cycles)
  - Executing a warp instruction = 4 cycles (for 8 scalar processors)

- So, to hide L1 cache latency: 28/4 = 7 warps
- To hide global memory latency: 350/4 = 88 warps

GPU Programming Model
Logical Execution

Frameworks

- **Open Computing Language (OpenCL)**
  - Vendor independent, industry standard
  - Different hardware (CPU, GPU, FPGA)
  - C/C++

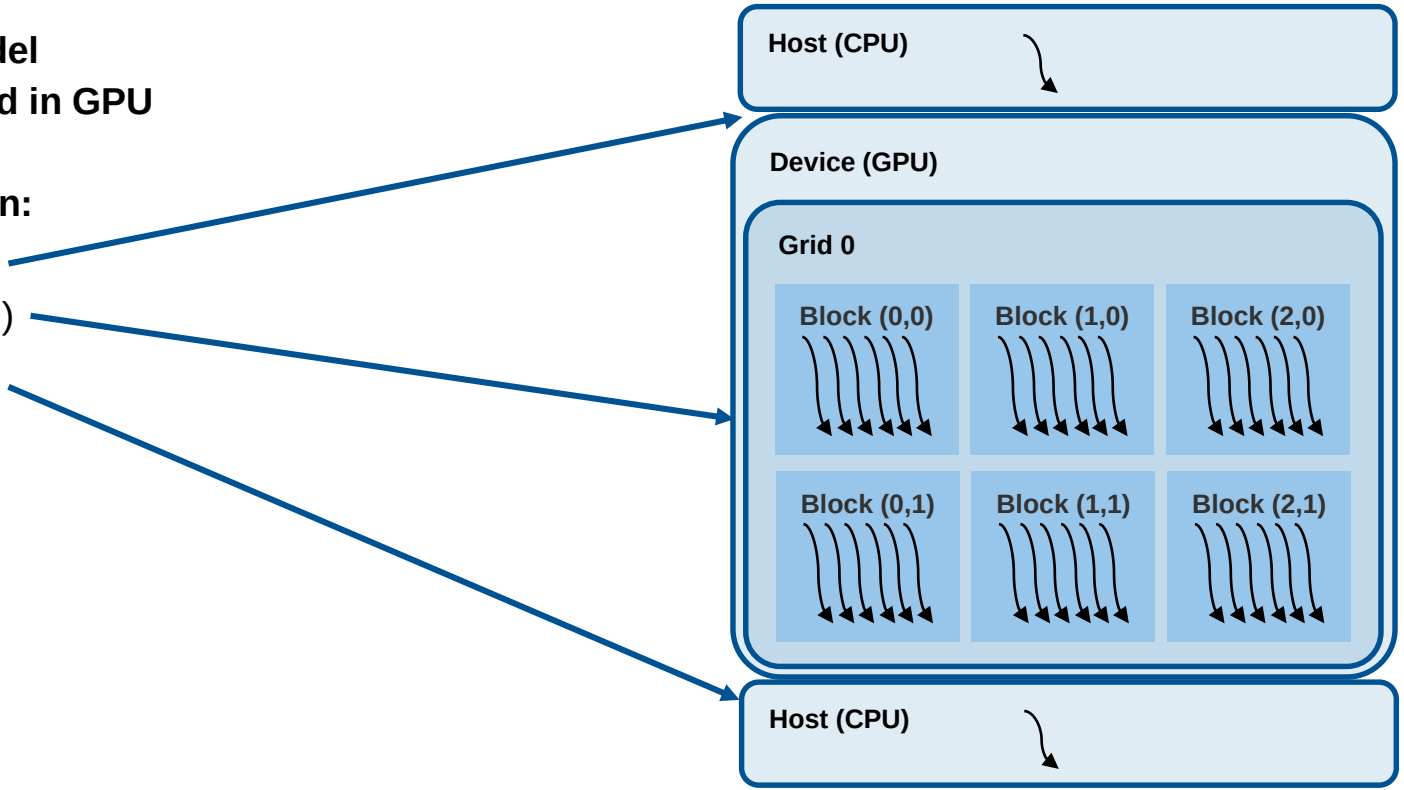
- **Compute Unified Device Architecture (CUDA)**
  - Proprietary for NVIDIA GPUs
  - C/C++, Fortran
  - Wrappers for Python, Perl, Mathematica, F#

- **AMD ROCm**
  - Open source, vendor independent
  - Different hardware (CPU, GPU, APU)

GPU Program Execution

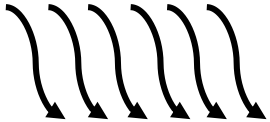
- **Heterogeneous model**
- **Operates in CPU and in GPU**

- **Sequential execution:**
  - Serial code (CPU)
  - Kernel mode (GPU)
  - Serial code (CPU)



Execution Flow

- **A kernel is executed by many GPU threads**
  - Kernel = GPU equivalent of a CPU thread



- **A GPU thread executes a sequence of instructions assigned to a computational core**

- **Achieving massive parallelism**
  - High number of threads are spawned at the same time
  - Single instruction multiple threads (SIMT) execution model

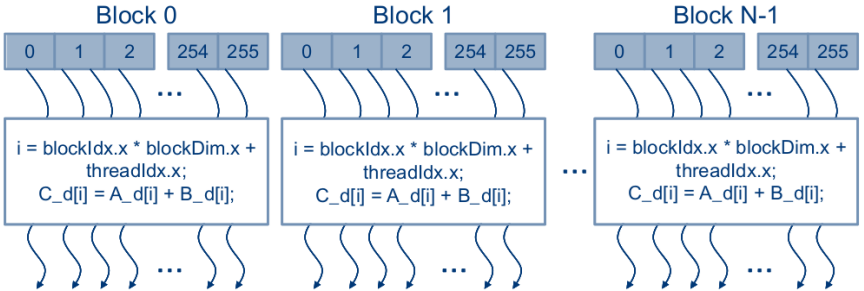
- **SIMT vs. SIMD**
  - SIMD – single instruction is executed on all data
  - SIMT – single instruction is executed only on the active threads

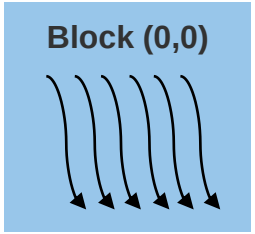
GPU Threads, Blocks and Grids

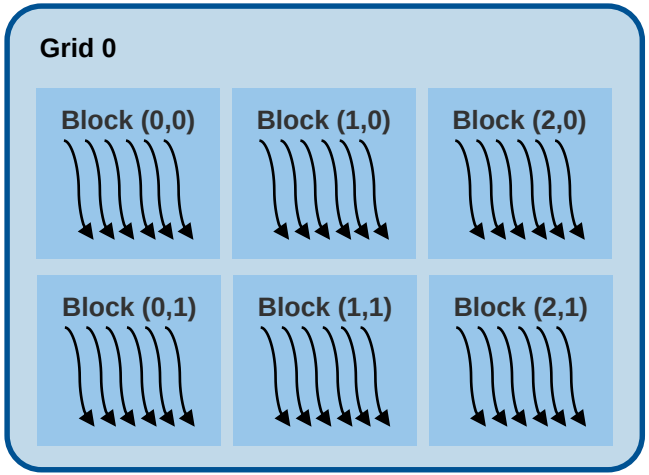
- **Threads are grouped into thread blocks**
  - All threads in one thread block run in the same SM
  - Threads in the same block can communicate

- Each threads has a **unique <blockID, threadID> pair**
- Can *explicitly* access **different parts of the data**

- **Blocks form a grid**
- Thread blocks are *independent*
- They are executed in random order





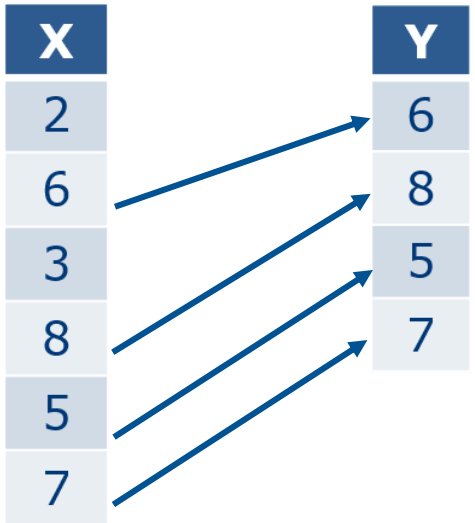


GPU acceleration in Databases

Selection

- Given a predicate P, choose a subset of tuples from a relation R that satisfy P and remove the rest.

P: X>4

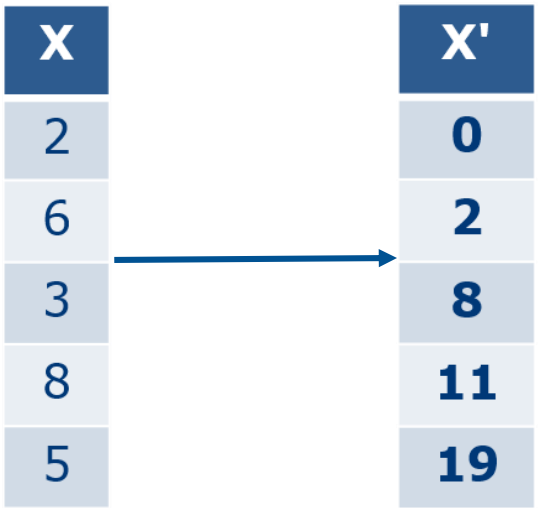



```
int i=0;
for (i=0; i<n; i++){
   if (pred(X[i])) {
       Y.add(X[i]);
   }
}
```

Prefix-scans

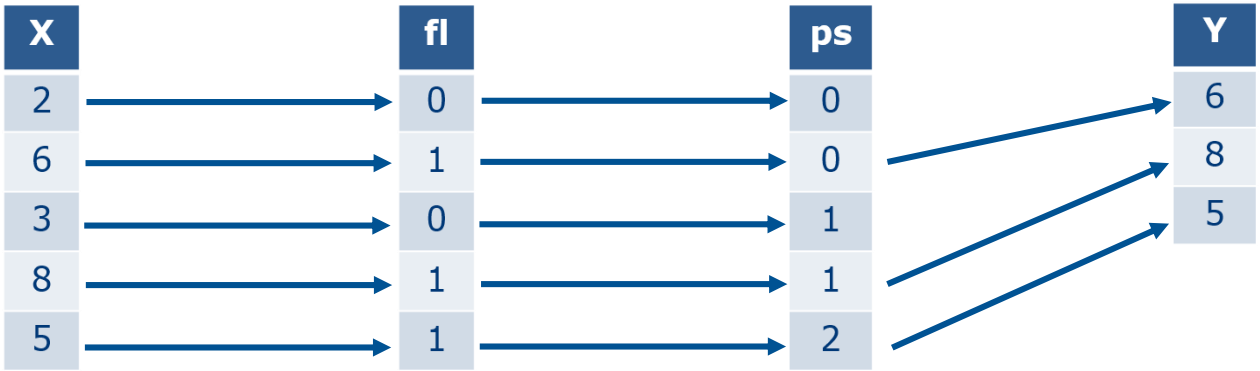
- **Apply a binary associative operator ⊕ to an array of n elements**.
  - Given an input a = [a0, a1, …, an-1]
  - The output is a'=[0, a0, (a0⊕a1), (a0⊕a1⊕a2), …, (a0⊕a1⊕…⊕an-2)]

Prefix sum:



Parallel selection

- Predicate P: X>4
- Idea: pre-compute locations by prefix sum



1. Build flag array ***fl***:

```
if(P(X[i])
    fl[i]=1;
else
    fl[i]=0;
```

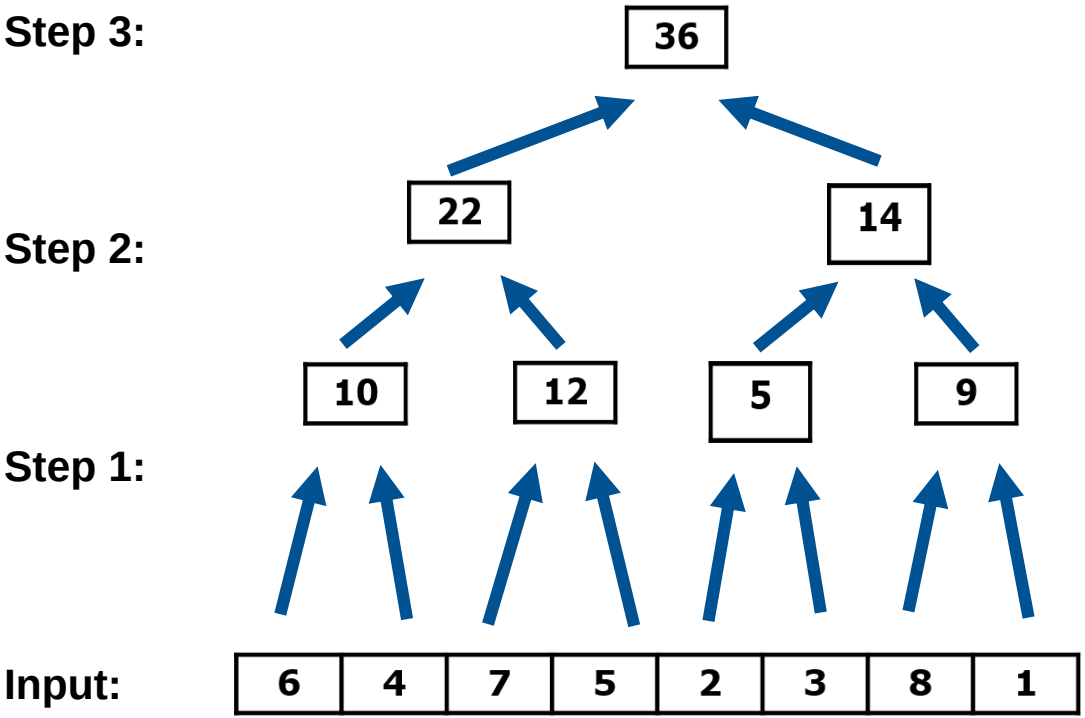
2. Compute prefix sum ***ps*** from ***fl***

3. Scan ***fl*** and write ***X[i]*** to ***ps[i]*** in ***Y***

Prefix sum: a reduction

- **Reduction: reduce a set of values to a single value using a binary operator ⊕**
- **From the prefix scan slide:**
  - Given an input a = [a0, a1, …, an-1]
  - The output is a'=[0, a0, (a0⊕a1), (a0⊕a1⊕a2), …, (a0⊕a1⊕…⊕an-2)]
- We can use reductions as a **building block** for a **prefix sum operation**
- Question: How can we parallelize **reductions** and **prefix scan operations** in general?

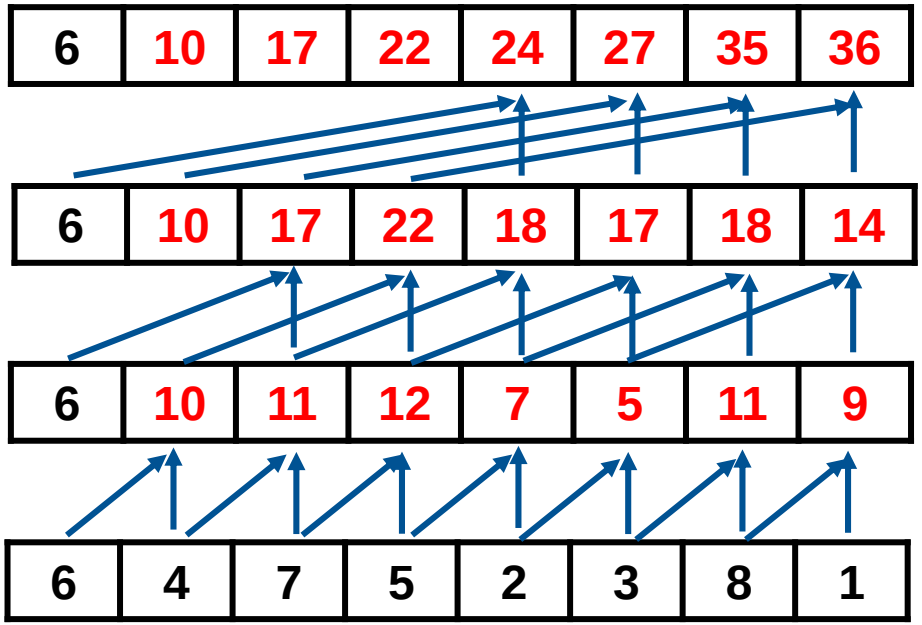
Binary reduction trees



- To compute the sum of an array, we require log(n) steps with a binary reduction tree.
- The input array can be split into X blocks of 2 elements to be summed.
- We assign each block of two elements to a single thread.

Reduction Trees on GPUs

- A naïve implementation based on Hillis and Steele (‘86), on GPUs by Horn(‘05)
- But, is not work efficient
  - It performs more add operations O(n log n) than a sequential implementation O(n)
- **Q: How can we improve it?**

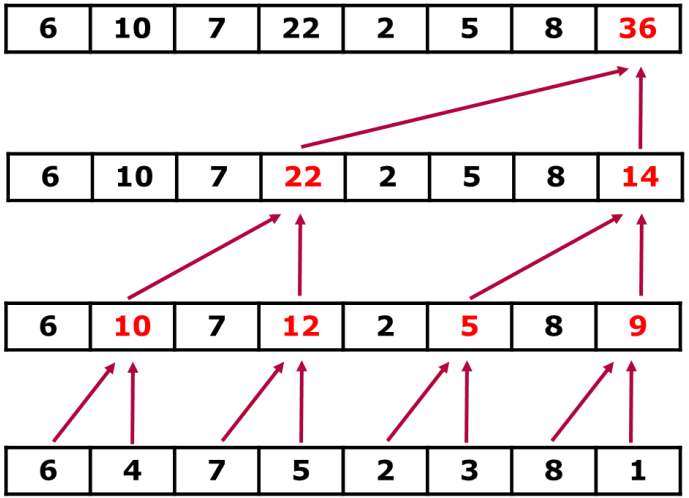


Reduction Trees on GPU

- Blelloch Algorithm (two phases, up-sweep shown here)
- One thread per two elements
- After every step, half of the threads remain active

- Single traversal of the tree performs (O(n)) add operations

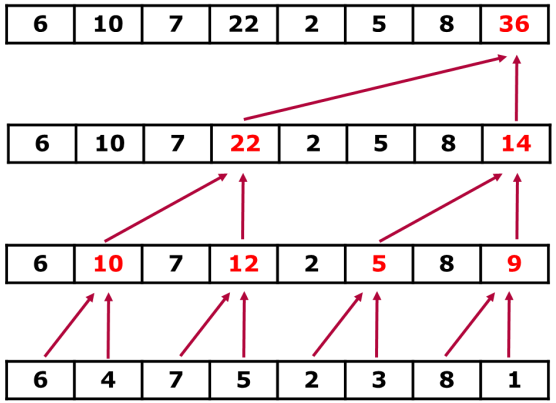
```
__global__ void SimpleSumReductionKernel (float* input, float* output){
    unsigned int i = 2*threadIdx.x;
    for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2){
        if (threadIdx.x % stride == 0){
            input[i] += input[i + stride];
        }
            __syncthreads();
    }
    if(threadIdx.x == 0){
        *output = input[0];
    }
}
```



- Strided memory access
- Each thread accesses 2 locations
- Stride doubles every iteration (1, 2, 4)
- **Is this efficient?**

Reduction sum kernel – Analysis

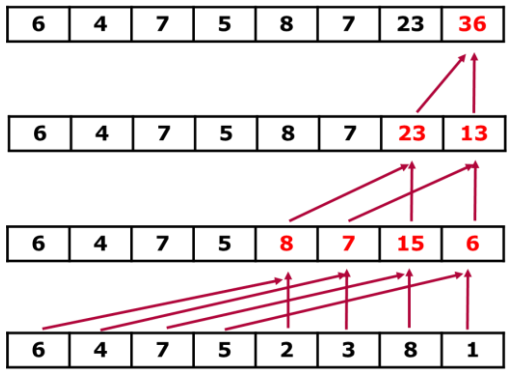
- In reduction kernels, we halve the number of active threads in each iteration
- Increasing the stride, increases the distance between the active threads
- After n iterations → underutilized warps
- Example: given an input of 256 elements
  - We activate 128 threads, i.e., 4 warps
  - We need log2(256) = 8 iterations to calculate the reduction sum
  - At iteration 6, only warps 0 and 2 are active with 1 thread each.
- **Question: how can we be more efficient?**



Technique: minimize control divergence

- Decrease the stride in each iteration
- So subsequent iterations operate on collocated threads and memory locations
- Adjacent warps and threads would deactivate
- Efficient use of the cache lines.

```
__global__ void ConvergentSumReductionKernel(float* input, float* output){
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >=1; stride /= 2){
        if (threadIdx.x < stride){
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0){
        *output = input[0];
    }
}
```



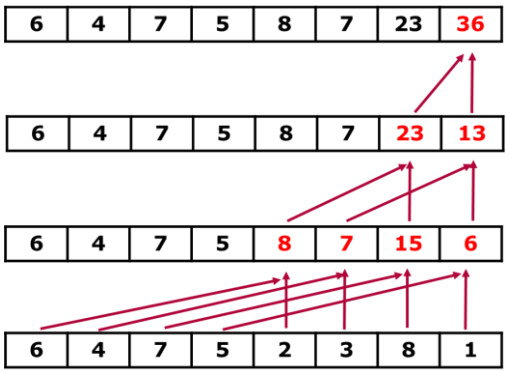
Technique: where you read/write matters

- Where do we write the intermediate results to?

- The input array is stored in global memory. → Load the input from global memory
- Partial results are written to global memory. → Write and read every intermediate from shared (shared access is 150x faster than global memory)

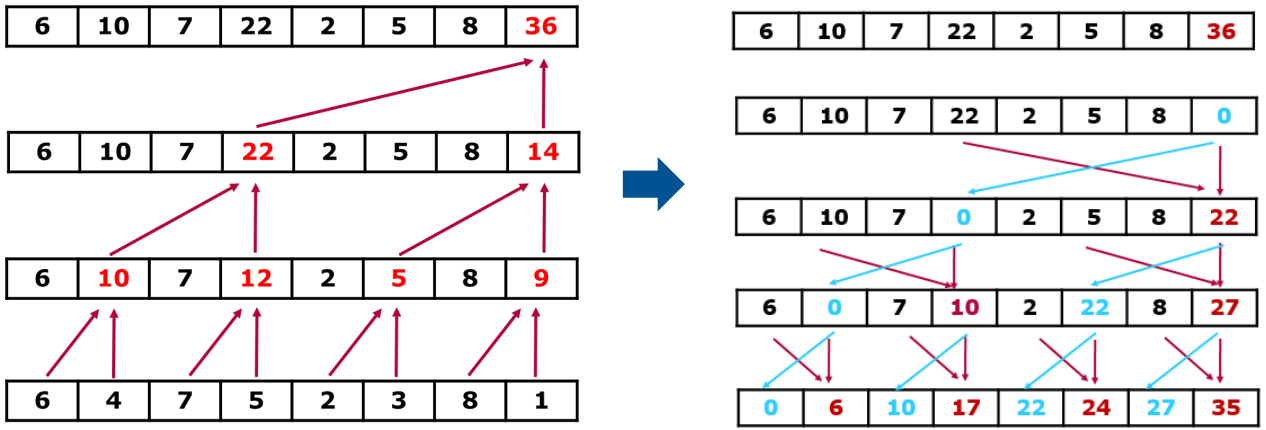
- Example: array with 256 elements
  - 36 global memory accesses/writes → Only initial input and final output are read/written, a total of 9 global memory accesses.

```
__global__ void ConvergentSumReductionKernel(float* input, float* output){
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >=1; stride /= 2){
        if (threadIdx.x < stride){
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0){
        *output = input[0];
    }
}
```



Reduction Trees on GPUs: final prefix sum

- With the current implementation, we do not have the partial sums yet.
- Solution:
  - Add one more phase – the sweep down phase
  - Add partial sums to successor elements
  - Output: final prefix sum



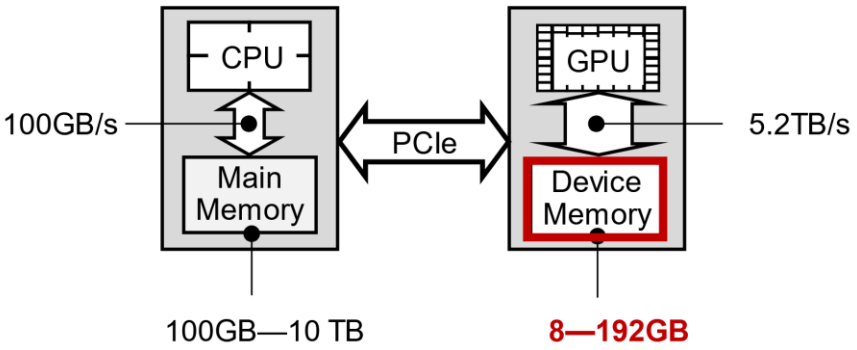
- Blue arrows: move down
- Red arrows: partial sums

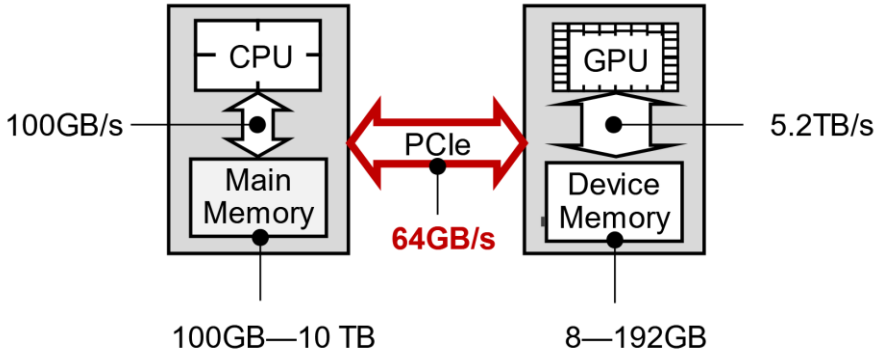
- Two phase parallel stage add operations: 2n

- Output: full prefix sum
- Same performance optimizations as earlier.

GPU considerations on DB-engine level

Challenges of GPUs for Data Analytics



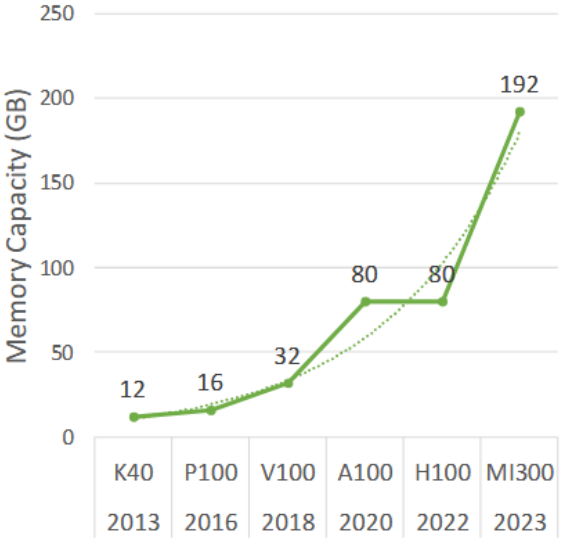


- **Challenge 1:** Limited memory capacity
  - Some data sets do not fit in GPU memory

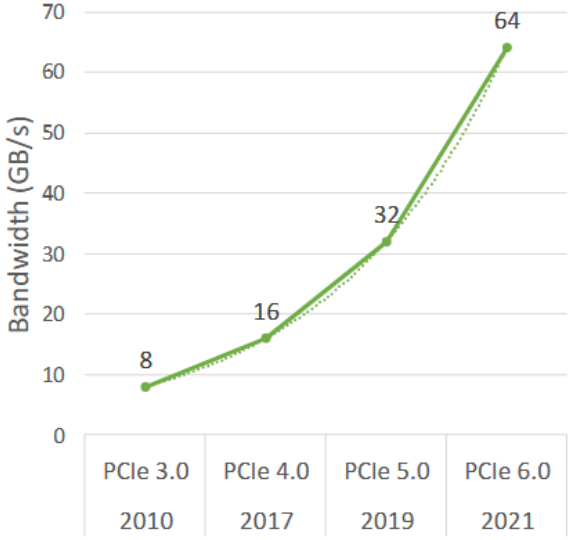
- **Challenge 2:** Limited interconnect bandwidth
  - Transferring data from CPU can be expensive

https://uwaterloo.ca/data-systems-group/sites/default/files/uploads/documents/talk-xiangyao-3-12.pdf

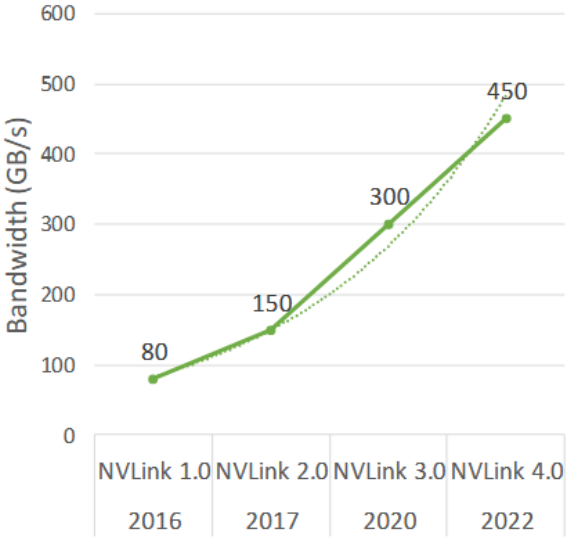
But, GPU trends…



(a) GPU Memory Capacity



(b) PCIe Bandwidth



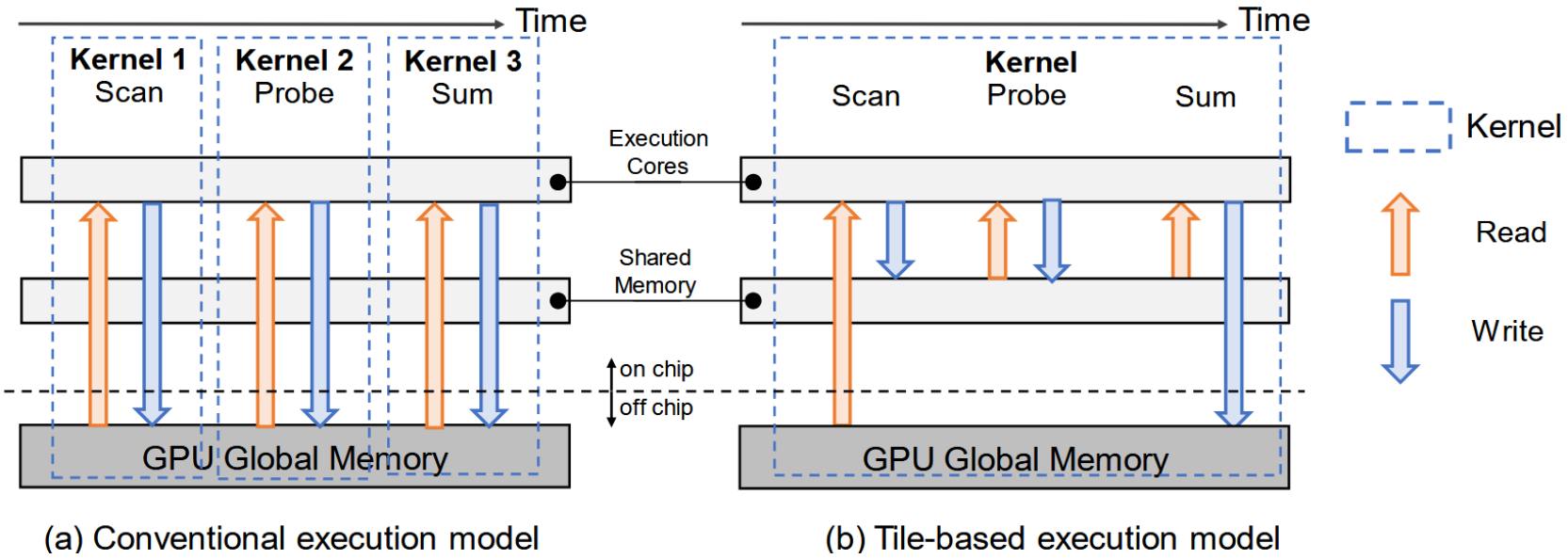
(c) NVLink Bandwidth

- GPU memory capacity increase by 6x in the last 5 years
- PCIe increases by 2x every two years
- NVLink bandwidth increase by 3x in 5 years.
  - NVLink C2C (2022) connects NVIDIA GPU and NVIDIA CPU (450 GB/s)

https://uwaterloo.ca/data-systems-group/sites/default/files/uploads/documents/talk-xiangyao-3-12.pdf

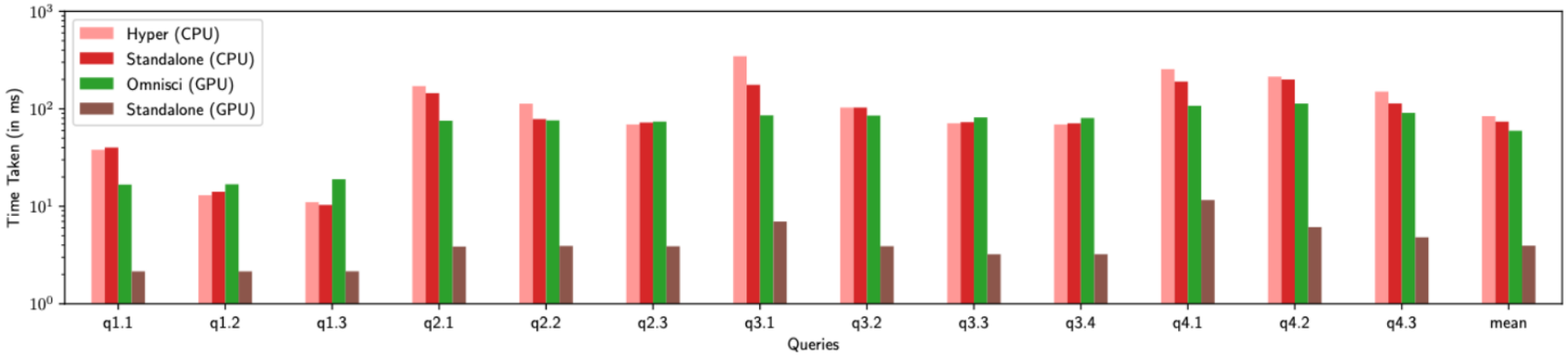
Tile-based execution model

- **Problem:** conventional execution model incurs *excessive memory traffic* for reading and writing intermediate results
- **Key idea:** partition data into *small tiles* and store intermediate results in the shared memory



https://uwaterloo.ca/data-systems-group/sites/default/files/uploads/documents/talk-xiangyao-3-12.pdf

How much does it help?



- With Crystal, GPU is on average 25x faster than CPU when running the Star-Schema-Benchmark (SSB)
  - Hardware: V100 GPU, Intel i7-6900 CPU (8 cores), SSB (SF 20)

https://uwaterloo.ca/data-systems-group/sites/default/files/uploads/documents/talk-xiangyao-3-12.pdf

Do we need to pre-load all the data?

- **Predicated loading primitive**
  - Keep track of valid tuples using a bitmap
  - Use the bitmap for on-demand tuple loading → avoid loading tuples that have already been invalidated by predicates on other columns

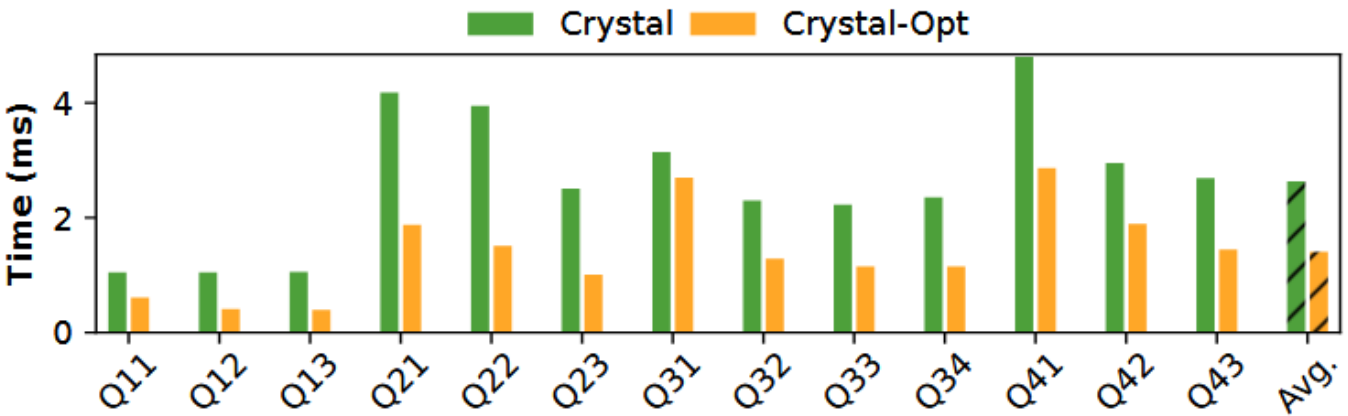
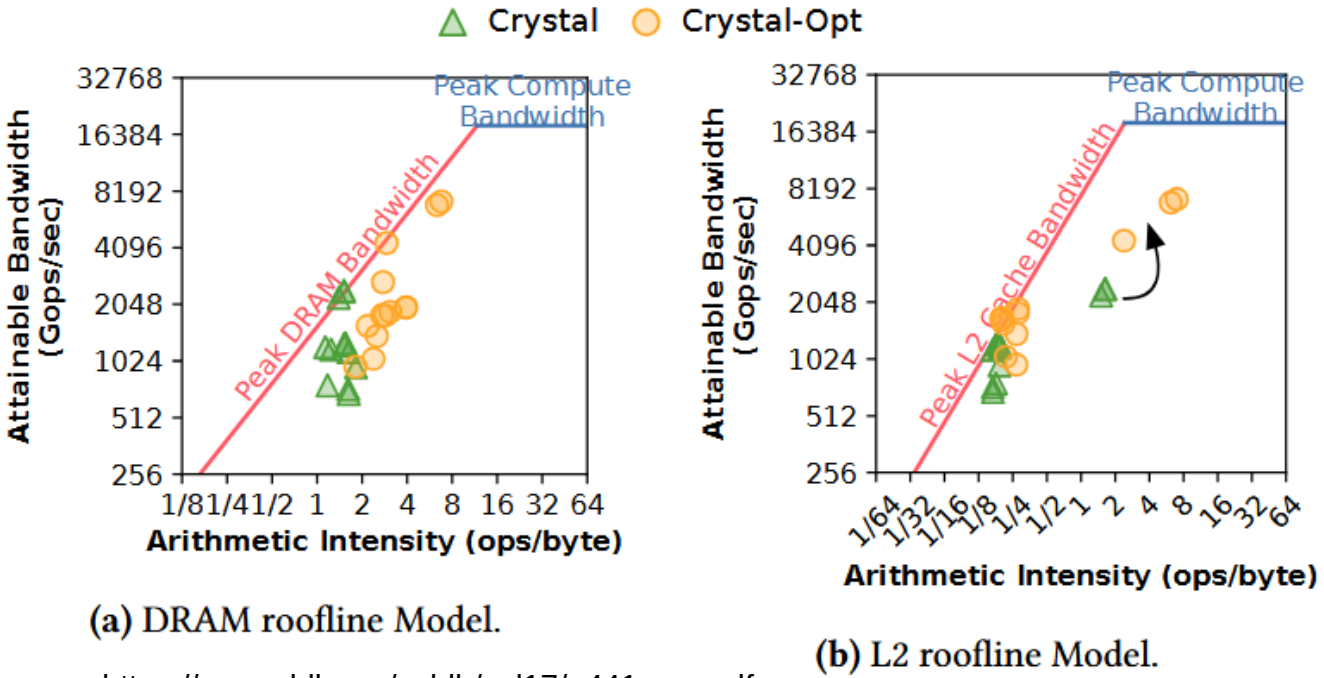


Figure 10: CRYSTAL vs CRYSTAL-Opt. query execution.

src: https://www.vldb.org/pvldb/vol17/p441-cao.pdf

What's the bottleneck?

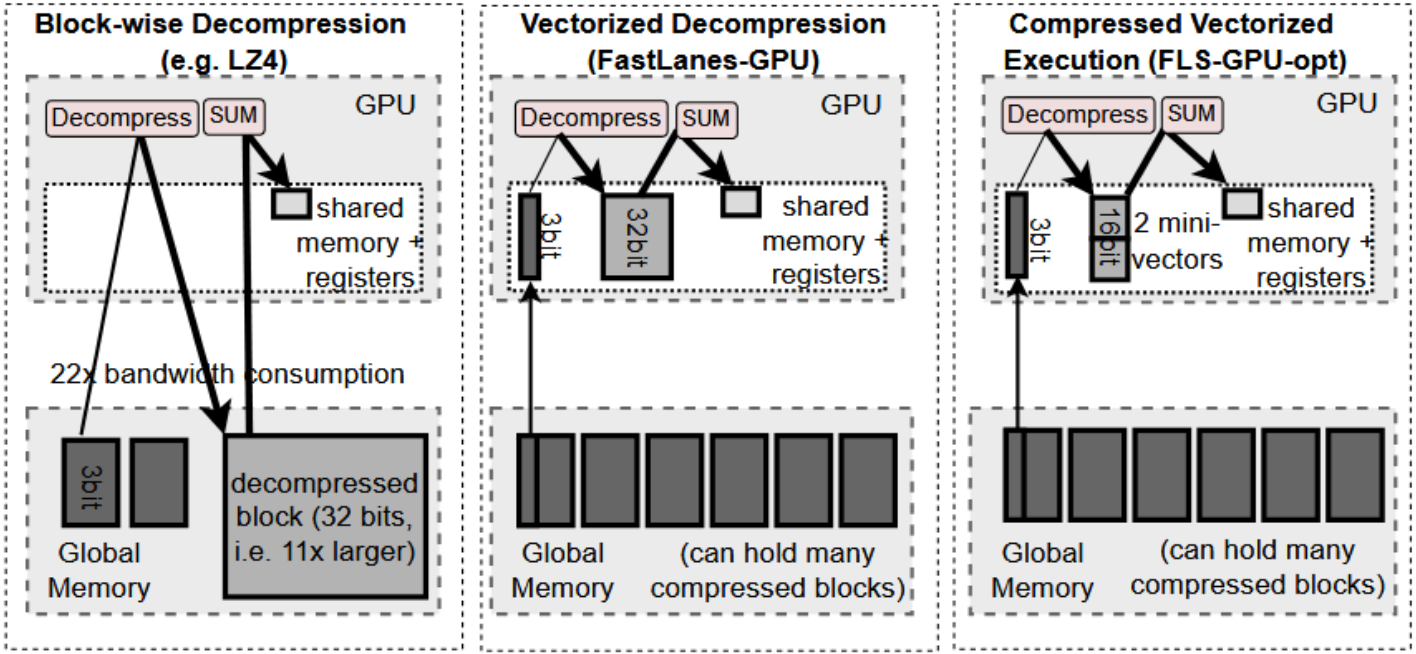
- Remember the Roofline model?



src: https://www.vldb.org/pvldb/vol17/p441-cao.pdf

Compression can help!

- Idea 1: working with tile-based compressed data
- Idea 2: GPU-optimized compression format, leverage the GPU registers and shared memory



https://dl.acm.org/doi/pdf/10.1145/3662010.3663450

Performance analysis on compression

- SSB Q1.1 is a simple scan with filter and aggregation.
- The roofline analysis above shows that it is the most scan (i.e., bandwidth) bound query.
- FLS-GPU: Interleaving of values in a vector employed by FastLanes → 2-3x faster
- -opt: predicate-pushdown brings additional benefit

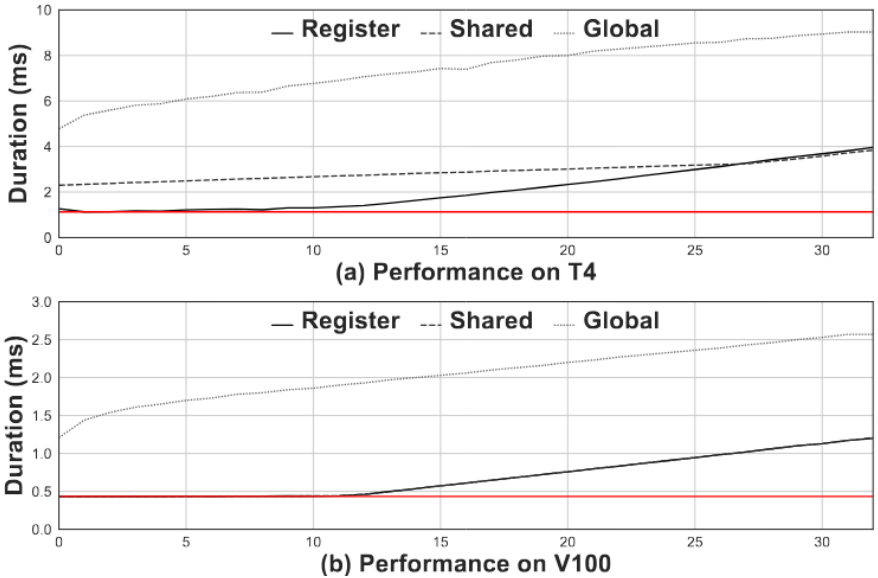


Table 11: On the scan-bound Q1.1 that stands to profit most from compressed scans, FLS-GPU shows strong performance, which is significantly enhanced in FLS-GPU-opt.

Scheme	SF1-T4	SF10-T4	SF1-V100	SF10-V100
Crystal	0.35	3.39	0.115	1.080
Crystal-opt	0.26	2.49	0.070	0.608
FLS-GPU	0.21	1.92	0.087	0.642
FLS-GPU-opt	0.139	1.19	0.057	0.335

https://dl.acm.org/doi/pdf/10.1145/3662010.3663450

References

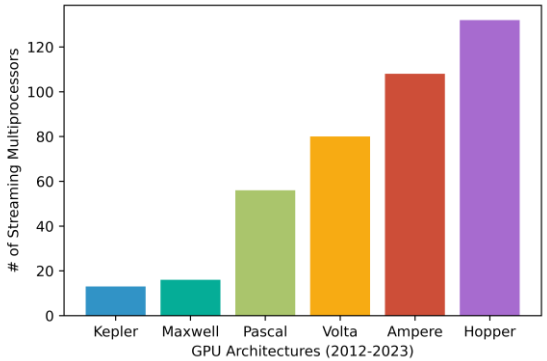
- **Lecture:** *Hardware-conscious Data Processing* by Prof. Tilmann Rabl (HPI)
- **Slides:** from Xiangyao Yu (University of Wisconsin Madison) on *GPU Databases – The New Modality of Data Analytics*
- **Blog:** by Mark Harris (NVIDIA), Shubhabrata Sengupta (UC Davis), and John Owens (UC Davis)
  https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

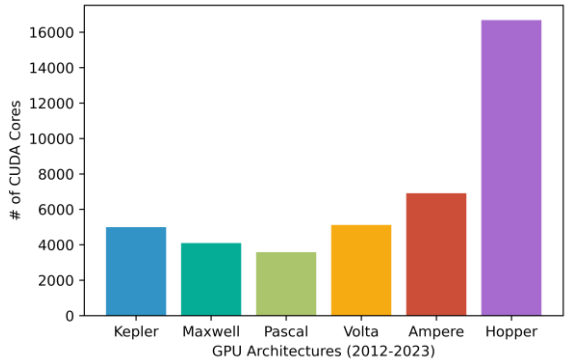
- **Book:**
  - David B. Kirk and Wen-mei W. Hwu
    "Programming Massively Parallel Processors" – a hands-on approach, 4th edition (2023)
    - chapter 10: Reduction (and minimizing divergence), chapter 11: Prefix sum (scan)

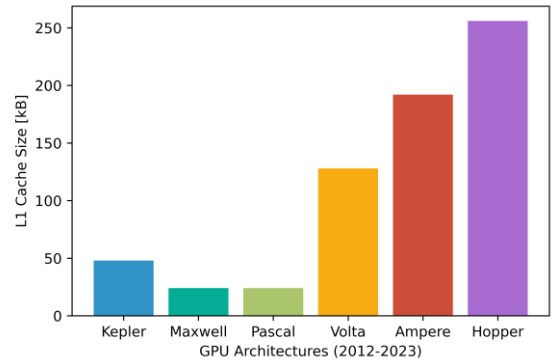
- **Various papers:**
  - Afroozeh et al. *"Accelerating GPU Data Processing using FastLanes Compression".* ACM DaMoN 2024
  - Cao et al. *GPU Database Systems Characterization and Optimization.* PVLDB 2023
  - Shanbhag et al, *A study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics.* SIGMOD 2020

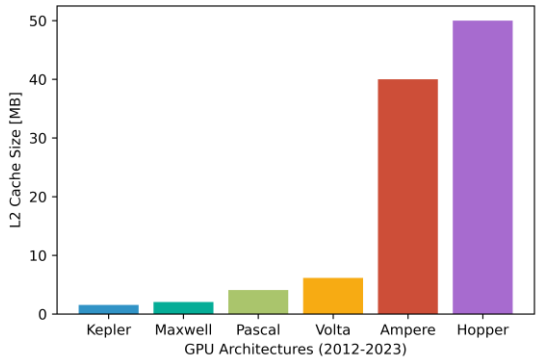
Appendix

Evolution of GPUs

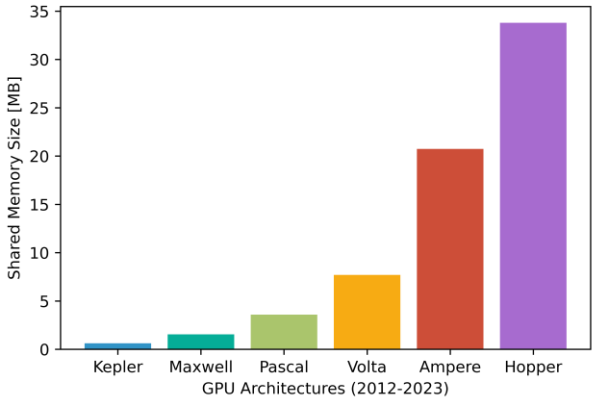


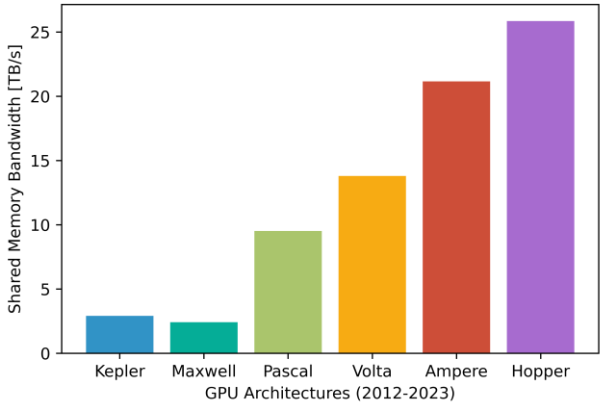


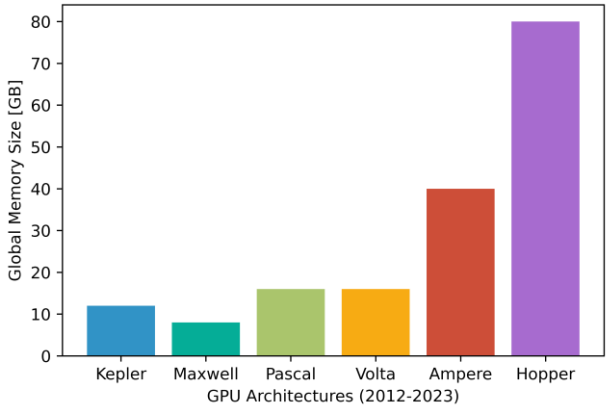


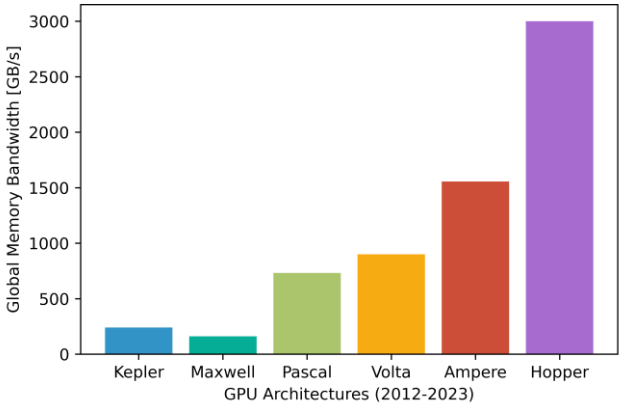


Evolution of GPUs II









Performance Considerations -- overview

Optimization	Compute Benefits	Memory Benefits	How To
Coalesced Memory Access	Fewer stalls for memory access	Better utilization of cache lines	Rearrange the mapping of threads to data. Rearranging the layout of the data.
Maximizing Occupancy	Hide pipeline latency	Hide DRAM latency	Tune SM usage: threads/block, shared memory/block, registers/thread.
Thread Coarsening	Less divergence and synchronization	Less global memory traffic	Assign multiple units of parallelism to each thread to reduce the price of parallelism.
Privatization	Fewer stalls for atomic updates	Less contention and serialization of atomic updates	Partial updates to a private copy of the data and updating the universal copy once when done.
Minimize Control Divergence	High SIMD efficiency	-	Rearrange the mapping of threads to data. Rearranging the layout of the data.
Tiling Reused Data	Fewer stalls for memory access	Less memory traffic	Place reused data in a block of shared memory.