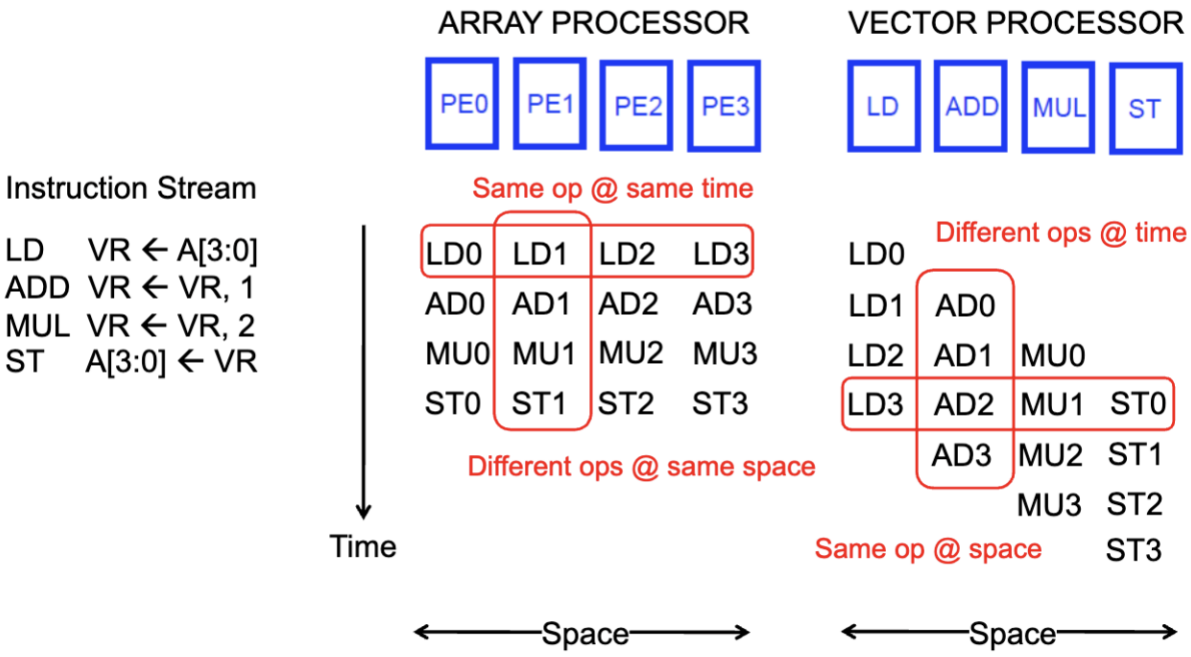
19_SIMD architecture

> https://www.youtube.com/watch?v=gkMaO3yJMz0

        - Example: SPMD on SIMT machine
        - How do they operate
    - Warp and Thread Management:
    - SIMD vs SIMT Execution Model
    - Advantages of SIMT
    - Fine-grained Multi-threading of Warps
    - High-level Overview of GPU Architecture
    - GPU vs. CPU
    - SIMD vs SIMT
        - Warp Instruction Level Parallelism
        - Memory Access in GPU Architecture
    - Warp Exposure to Programmers
        - From Blocks to Warps in GPUs
        - Warp-based SIMD vs. Traditional SIMD
    - SPMD: Single Program (/ Procedure) Multiple Data
    - Flexibility of SIMT in Grouping Threads into Warps
        - Conditional control flow instructions
        - Control flow problem in GPUs / SIMT
        - Dynamic Warp Formation / Merging
    - When are GPUs not efficient?
        - Example of a GPU - NVIDIA GeForce GTX 285
        - Tensor Cores
    - SIMD vs SIMT Summary
    - Limits of SIMD Architecture
    - Limits of SIMT Architecture
        - Comparison and Context
- 21_GPU_PROGRAMMING
    - Topics
    - Tensor cores
    - Terminology
    - Why CUDA (GPUs) is better than SIMD
        - Drawbacks
    - CPU vs GPU
        - CPU
        - GPU
    - GPU computing
        - Traditional program structure
            - Terminology
            - Recall: SPMD
        - CUDA / OPENCL Programming Model
            - Traditional program structure in CUDA
            - Indexing and memory access
                - How is it actually stored in memory?
            - 1D Grid (One way to do it)
            - 2D Grid (Another way to do it)
        - Review of GPU Architecture

    - Workflow
    - CUDA runtime

- **SIMD (Single Instruction, Multiple Data)** architecture is a type of parallel computing architecture that performs the same operation on multiple data points simultaneously.
- In a SIMD architecture, multiple processing elements execute the same operation on different pieces of data at the same time, thereby achieving data-level parallelism.
- Two common forms of SIMD architectures are **vector processors** and **array processors**.
    - **Array Processors**: In array processors, instructions operate on multiple data elements simultaneously using different processing elements. This form of SIMD architecture is characterized by having multiple processing elements that perform the same operation at the same time across different sets of data.
    - **Vector Processors**: Vector processors use a single instruction to perform operations on data stored in vector registers. These registers can hold multiple data elements, and a single instruction can operate on all these elements sequentially or in parallel, depending on the architecture.



- A **GPU** represents a hybrid of array and vector processor architectures.
- The **SIMD (Single Instruction, Multiple Data) architecture** utilizes registers that store vectors, which are arrays of N elements of M bits each, instead of single scalar values.

SIMD vs Out-of-Order Execution:

SIMD

Pros:

- **Parallel Data Processing**: SIMD allows a single instruction to simultaneously perform the same operation on multiple data points. This is particularly beneficial for tasks that can be parallelized at the data level, such as vector and matrix operations common in graphics, multimedia processing, and scientific computations.
- **Efficiency**: By executing one instruction across many data elements, SIMD reduces the instruction cycle overhead, leading to better utilization of processing power for suitable tasks.

- **Energy Efficiency**: Executing multiple operations simultaneously can be more energy-efficient than processing them serially, which is particularly valuable in mobile and embedded applications.

Cons:

- **Specialized Use Cases**: SIMD is most effective when operations are uniform and can be applied simultaneously to multiple pieces of data. It is less useful for general-purpose computing where each data element might require different operations.
- **Programming Complexity**: Utilizing SIMD often requires explicit programming effort, including managing data alignment and handling cases where data sizes do not match SIMD register widths.

Out-of-Order Execution

Pros:

- **Increased CPU Utilization**: Out-of-order execution allows CPUs to make more efficient use of processor cycles by executing instructions as resources become available, rather than adhering strictly to the program order.
- **Latency Hiding**: It helps in hiding the latency of slower operations like memory fetch and long arithmetic operations by rearranging the execution order to keep the execution units busy.
- **General Purpose**: This approach is beneficial for a wide range of applications, as it automatically optimizes execution without special programming requirements.

Cons:

- **Complexity**: The CPU design becomes significantly more complex with out-of-order execution, which can increase the cost and power consumption of the processor.
- **Diminishing Returns**: For highly sequential code or when dependencies between instructions are too tight, the advantages of out-of-order execution can be limited.

Comparison

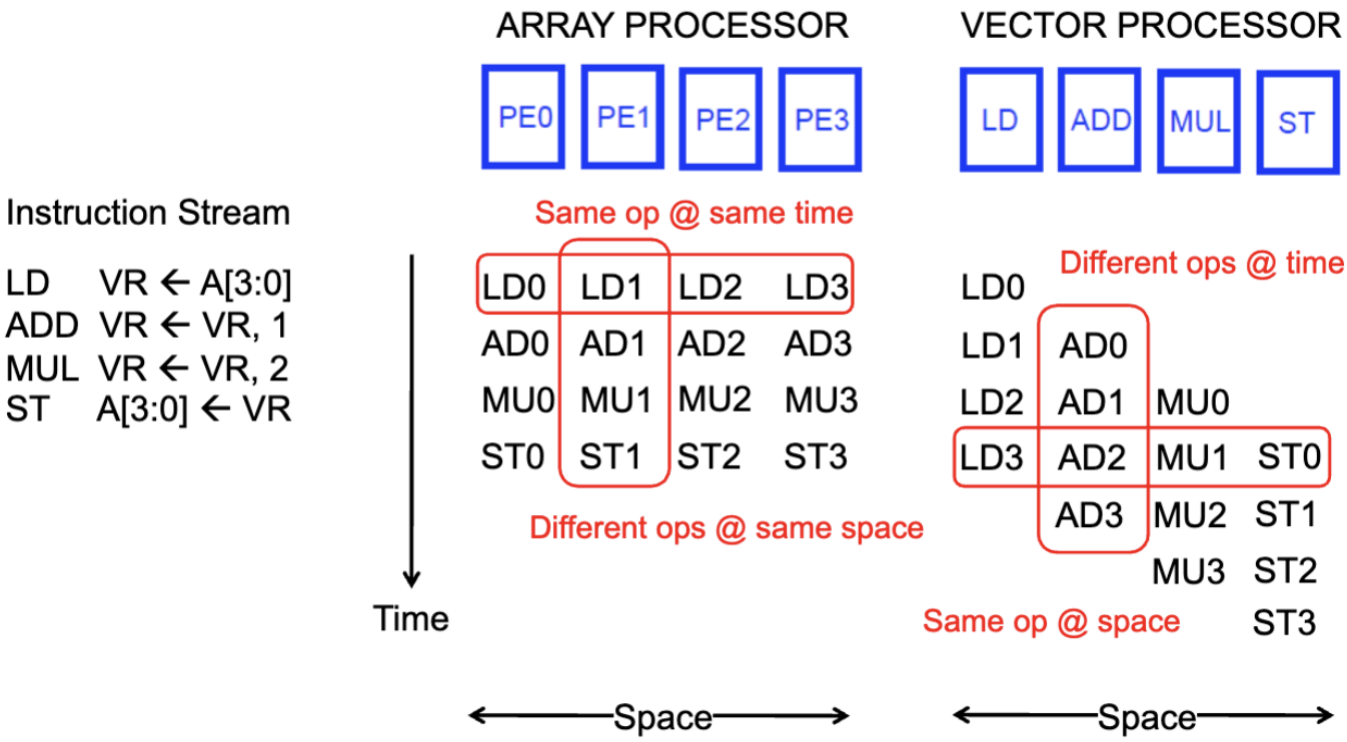
- SIMD and out-of-order execution are complementary techniques that address different aspects of parallelism and efficiency in computing.
- SIMD is well-suited for tasks with data-level parallelism, while out-of-order execution is more effective for improving the utilization of CPU resources and handling complex instruction dependencies.

Data Processing in SIMD Architectures:

- **Vector processors** handle data serially from vector registers:
    - Each cycle processes one element from the vector through a functional unit (FU) until all vector elements are processed.
- **Array processors** handle entire vectors simultaneously:
    - Performs operations like LOAD, ADD, and STORE on all elements at the same time, necessitating multiple FUs.

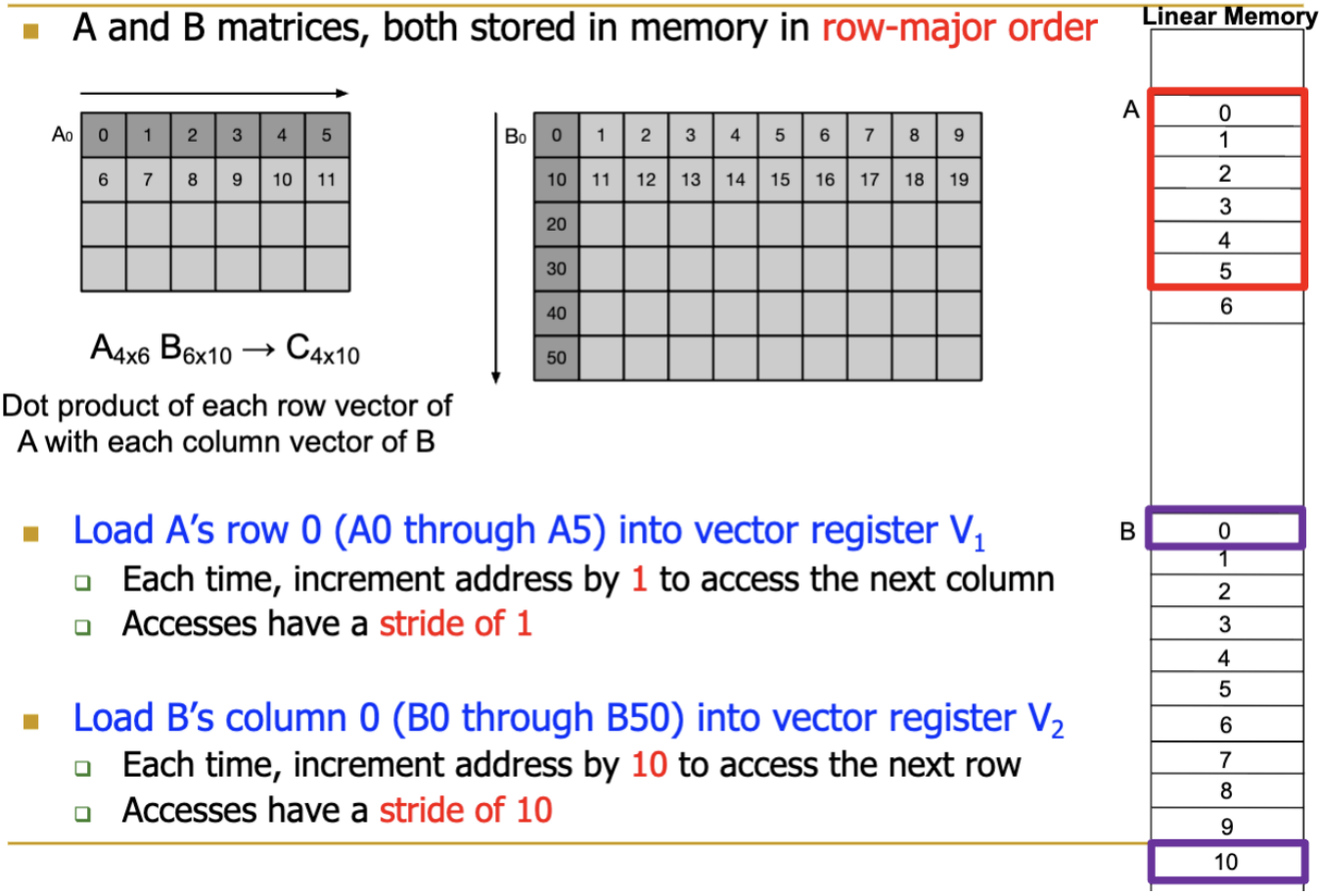
Example of Operational Differences:

- **To add two vectors of 32 elements:**
  - An **array processor** would perform 32 LOADs, 32 ADDs, and 32 STOREs concurrently, requiring 32 FUs.
  - A **vector processor** would sequentially perform:
    - LOAD0, then LOAD1 and ADD0, followed by LOAD2, ADD1, and STORE0, continuing in this fashion, utilizing fewer resources but at a slower rate.



Vector Stride and Pipeline:

- **Vector stride** refers to the memory distance between consecutive elements in a vector. A stride of 1, where elements are contiguous, is optimal for performance.
- Vector instructions benefit from longer pipelines due to:
  - Lack of data dependencies, eliminating dependency checks.
  - Absence of pipeline interlocks and control flows between vector elements.
  - Predictable strides that simplify data prefetching and caching.



Advantages of Vector Processors:

- **High Data Throughput**: Processes multiple data elements simultaneously, enhancing performance for data-intensive tasks.
- **Efficient Use of Resources**: Amortizes instruction decoding overhead across multiple data elements, reducing the total number of instructions and saving processing power.
    - Reduces instruction fetch bandwidth requirements
    - Amortizes instruction fetch & control overhead over many data -> leads to high energy efficiency per operation
    - Fewer loops & therefore fewer branches
- **Reduced Power Consumption**: Fewer instructions and efficient parallel processing lower energy usage, beneficial in energy-sensitive environments.
- **Simplified Programming for Parallelism**: Inherent hardware parallelism simplifies coding for applications suited to vector operations, like matrix computations.
- **Optimized Performance for Specific Applications**: Excels in applications involving large vectors or matrices, significantly outperforming scalar processors in these cases.
- **Pipelining**: Employs deep pipelining to process different stages of vector operations simultaneously, boosting instruction throughput.
    - No intra-vector dependencies -> no hardware interlocking -> no pipeline stalls to prevent hazards.
    - No control flow within a vector
    - Predictable strides for efficient data prefetching and caching.
- **Scalability**: Scales effectively with added hardware resources, handling larger vectors or more operations without major power or complexity increases.

- **Handling Large Data Sets**: Ideal for modern applications involving large datasets, such as machine learning and big data analysis.
    - predictable memory access pattern

Disadvantages of Vector Processors:

- Dependence on parallelizable data, suitable only for regular parallelism.
    - Becomes very inefficient if parallelism is irregular
- Ineffective for data structures like linked lists, where the next element's position is unpredictable.
- Memory bandwidth can limit performance due to the large volume of data accessed per instruction.

To quote:

> To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area.

Amdahl's Law

- Amdahl's Law is a fundamental principle in parallel computing that quantifies the potential speedup achievable by parallelizing a computation.
- It states that the overall speedup of a program is limited by the fraction of the program that cannot be parallelized.
- The law is expressed as a formula:
    - **Speedup = 1 / [(1 - P) + (P / N)]**
    - Where:
        - **P** is the fraction of the program that can be parallelized.
        - **N** is the number of processors or cores available for parallel execution.

Vector Registers and Functional Units:

- **Vector registers** store multiple M-bit elements:
    - Controlled by **vector control registers**:
        - **VLEN** (vector length)
        - **VSTR** (vector stride)
        - **VMASK** (mask for conditional operations, enabling selective processing based on specified conditions, such as VMASK[i] = (V_k[i] == 0)).
- **Vector functional units (FU)** can be deeply pipelined, exploiting the independent processing of elements to enhance throughput.

Loading / Storing vectors from & to memory

- Multiple elements need to be loaded/stored from/to memory in a single instruction.
- The elements are separated by a fixed stride.

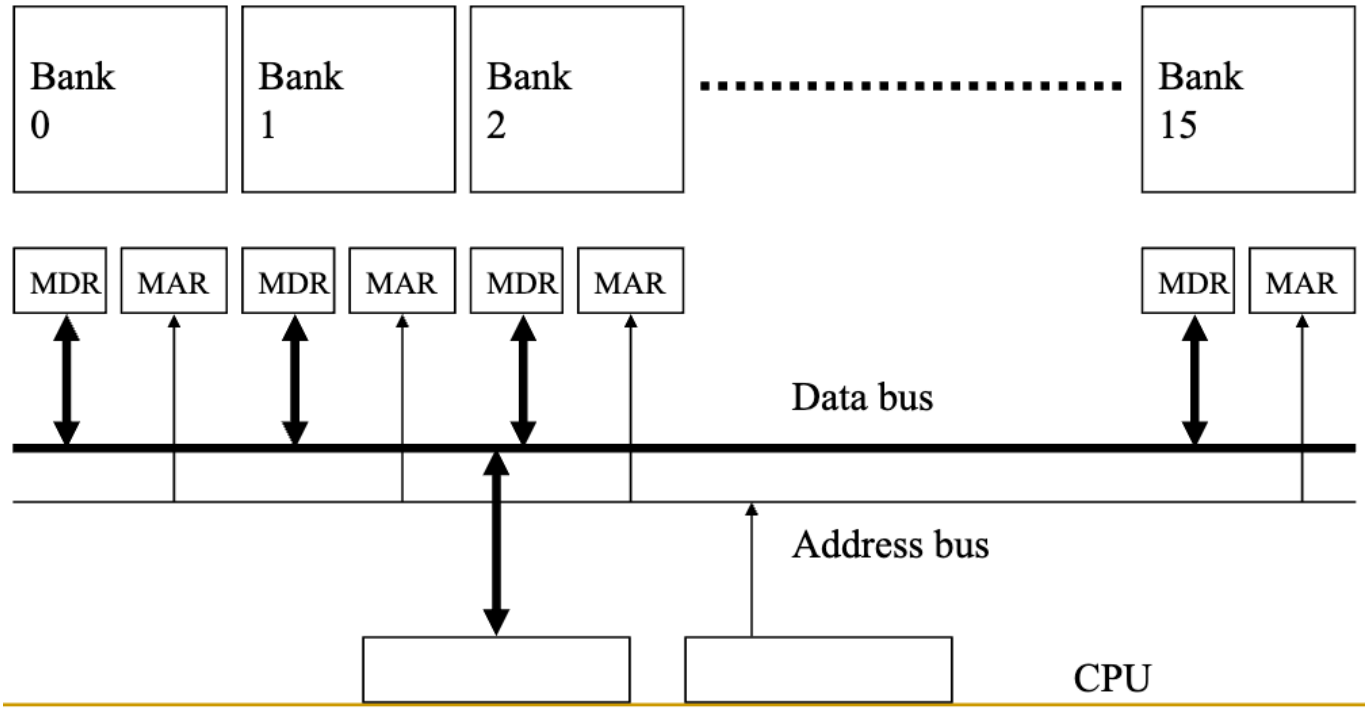
How do we achieve this with a memory that takes more than 1 cycle to access?

- Bank the memory

- Interleave the memory accesses across the banks

Memory banking

- Memory is divided into multiple banks, that can be accessed simultaneously.
- Banks share address and data buses (to reduce memory chip pins required)
- This way can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks



An example

- For i = 0 to 49
- C[i] = (A[i] + B[i]) / 2

```
    MOVI R0 = 50            1
    MOVA R1 = A             1
    MOVA R2 = B             1
    MOVA R3 = C             1
X:
    LD R4 = MEM[R1++]       11  ;autoincrement addressing
    LD R5 = MEM[R2++]       11
    ADD R6 = R4 + R5        4
    SHFR R7 = R6 >> 1       1
    ST MEM[R3++] = R7       11
    DECBNZ R0, X            2   ;decrement and branch if NZ
```

-> 304 dynamic instructions

- Assuming scalar execution time in-order with 1 bank

    - first 2 loads in the loop cannot be pipelined -> 2*11 cycles
    - 4+50*40 = 2004 cycles

- Assuming scalar execution time in-order processor with 1 bank with 2 memory ports (which can be used concurrently) or 2 banks

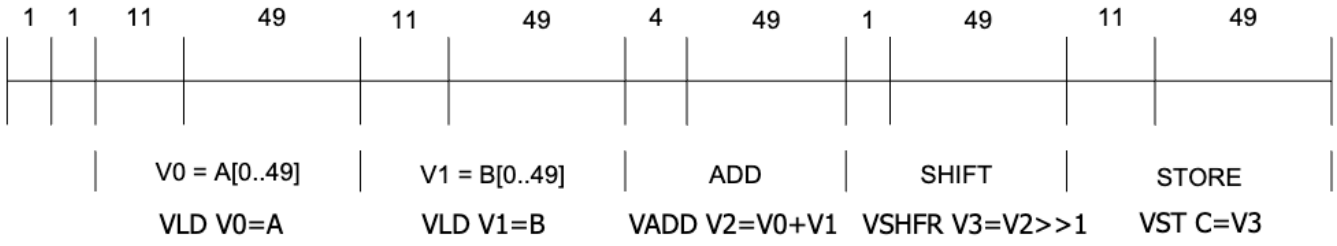
    - first 2 loads in the loop can be pipelined -> 1 + 11 cycles
    - 4+50*30 = 1504 cycles

- Since the loop can be vectorized (since each iteration is independent of any other)

```
MOVI VLEN = 50              1
MOVI VSTR = 1               1
VLD V0 = A                  11 + VLEN – 1
VLD V1 = B                  11 + VLEN – 1
VADD V2 = V0 + V1           4 + VLEN – 1
VSHFR V3 = V2 >> 1          1 + VLEN – 1
VST C = V3                  11 + VLEN – 1
```

- Assuming
    - **no chaining** (no vector data forwarding -> output of a vector FU cannot be used as input to another vector FU in the same cycle)
    - entire vector registers need to be ready before the operation can start
    - 1 memory port (one address generator) per bank
    - 16 banks (word-interleaved: consecutive elements of an array are stored in consecutive banks)

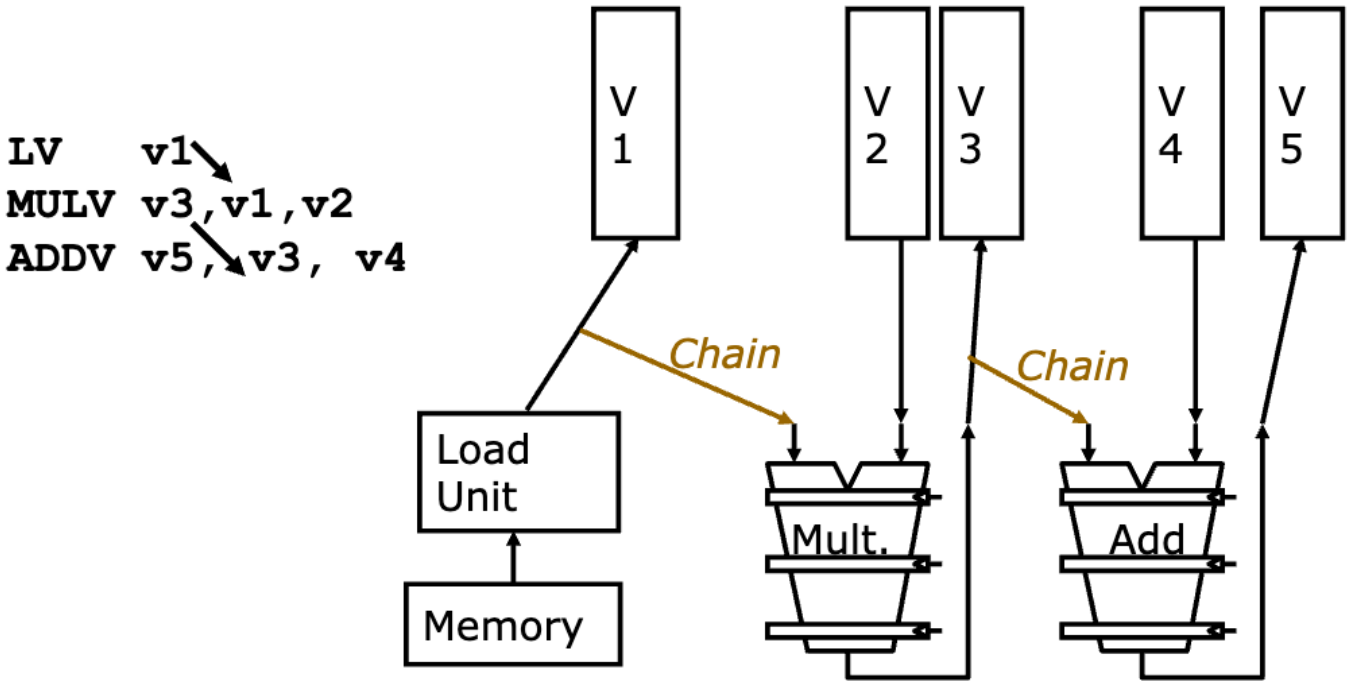
-> 1+1+11+49+11+49+4+49+1+49+11+49 = 285 cycles



- why 16 banks?
    - 11-cycle memory access latency
    - having 16 (> 11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency
- assuming unit stride of 1 (VSTR = 1)
- what if stride > 1?
    - how can we endure we can access 1 element per cycle when memory latency is 11 cycles? **VECTOR CHAINING**

Vector Chaining

- **Vector chaining** allows the output of one vector operation to be used as the input for the next operation in the same cycle.



- In the previous example, assuming vector chaining is possible
  - 1+1+11+49+11+49+49+11 = 182 cycles

Vector stripmining

- Occurs, for example, when # data elements > # elements in a vector register
- The loop can be broken down into # elements in the vector register
  - E.g. 527 data elements, 64-element VREGs
  - 8 iteration where VLEN = 64
  - 1 iteration where VLEN = 15 (the VLEN must be changed at runtime)

Gather / Scatter operations

- Occurs when the vector data is not stored in a strided fashion (irregular memory access to a vector)
- Indirection will be used to combine / pack elements into vector registers

Gather example

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

```
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base <- **Gathering**
LV vB, rB       # Load B vector
ADDV.D vA,vB,vC # Do add
SV vA, rA       # Store result
```

Scatter example

```
Index Vector                Data Vector (to Store)              Stored
vector (in memory)
0                           3.14                                Base+0
3.14
2                           6.5                                 Base+1  X
6                           71.2                                Base+2
6.5
7                           2.71                                Base+3  X
                                                                Base+4  X
                                                                Base+5  X
                                                                Base+6
71.2
                                                                Base+7
2.71
```

Conditional operations

- **Masking** is used to conditionally execute operations on vector elements
- VMask register is a bit mask determining which data element should not be acted upon

1 Example

```
for (i=0; i<N; i++)
    if (a[i] != 0) then b[i]=a[i]*b[i]
```

```
VLD V0 = A
VLD V1 = B
VMASK = (V0 != 0)
VMUL V1 = V0 * V1
VST B = V1
```

- Named, predicated execution in some architectures. Execution is predicated on mask bit

2 Example

```
for (i = 0; i < 64; ++i)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]
```

```

```

```
A       B       VMASK
1       2       0
2       2       1
3       2       1
4       10      0
-5      -4      0
0       -3      1
6       5       1
-7      -8      1
```

Implementation of masked operations

Simple

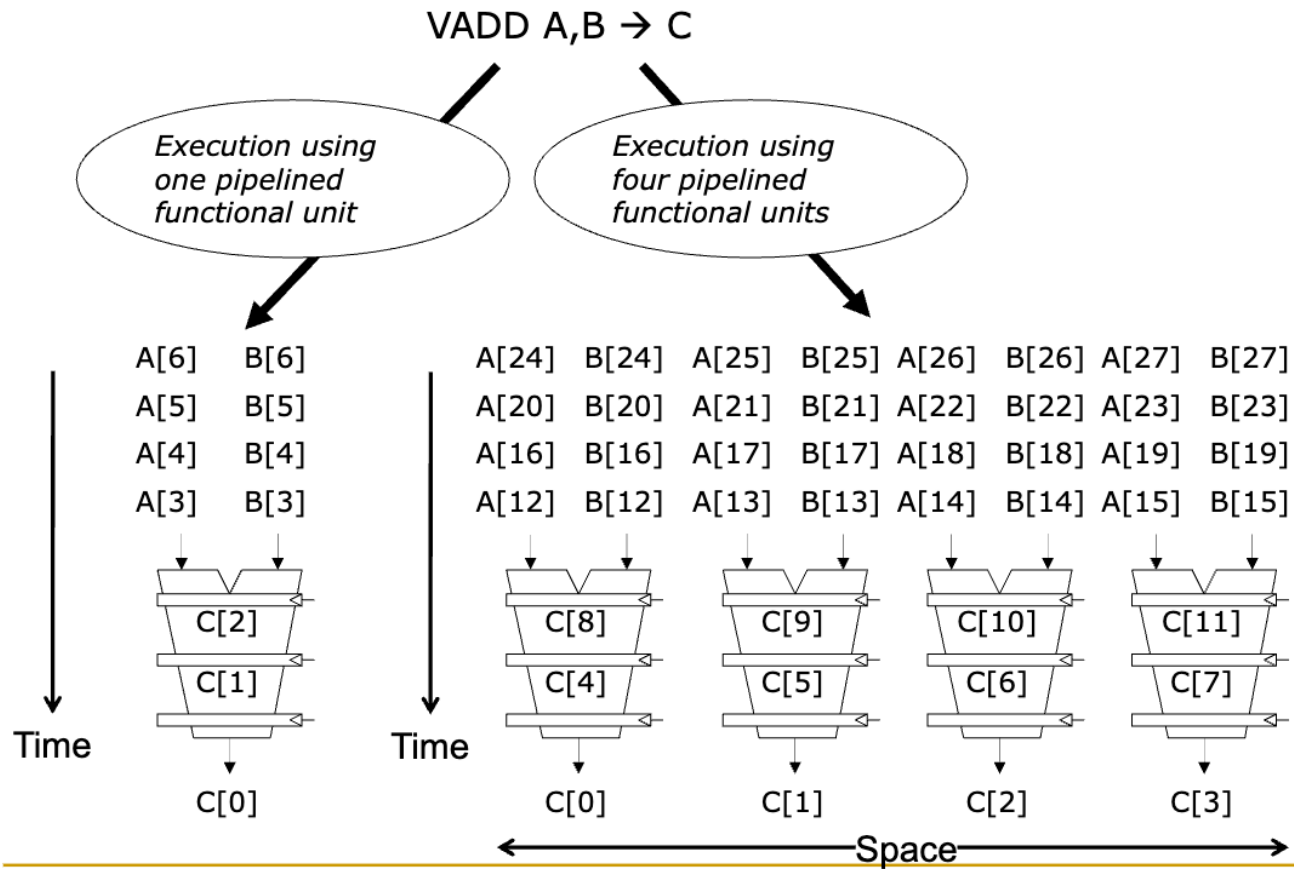
- Execute all N operations, turn off result WB according to the mask

Density-Time

- scan mask vector and only execute elements with non-zero masks

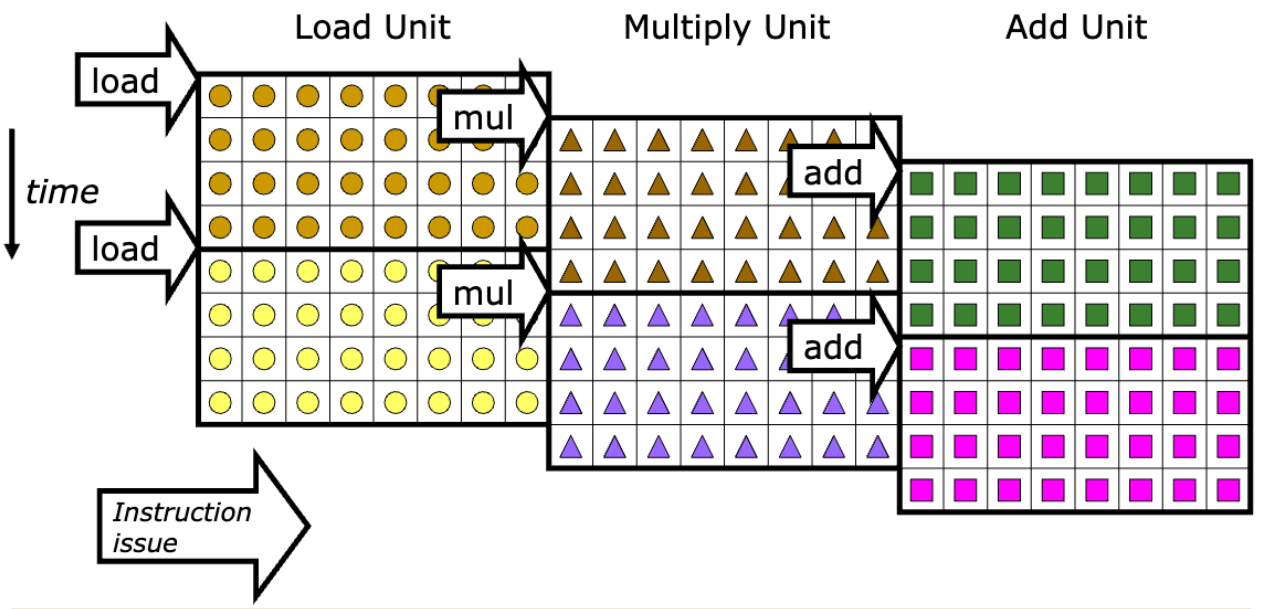
Array vs Vector processors, again

- Array vs vector processors distinction is a "purist's" distinction
- Most "modern" SIMD processors are hybrids of the two
  - exploiting data parallelism in both time and space
  - GPUs are a prime example



Vector instruction level parallelism

- Can overlap execution of multiple vector instructions
    - Example, machine has 32 elements per vector register and 8 lanes
    - Example, with 24 operations/cycle (on steady state) while issuing 1 vector instruction/cycle

> What is a lane? Refer to the individual execution units within a processor that can independently execute operations on different elements of a vector simultaneously. Each lane operates on a separate data element but under the same instruction.
>
> 
>

In summary

- Vector / SIMD machines are good at exploiting regular data-level parallelism
    - Same operation performed on many data elements
    - Improve performance, simplify design (no intra-vector dependencies)
- Performance improvement limited by vectorizability of code
    - Scalar operations limit vector machine performance
    - Amdahl's law
- Many existing ISAs include (vector-like) SIMD operations
    - Intel MMX / SSEn / AVX, PowerPC AltiVec, ARM Advanced SIMD (NEON)

20_GPU architecture

> https://www.youtube.com/watch?v=UFD8K-lprbQ

Overview of GPU Architecture

- GPUs are both vector processors and array processors, a mix of the two. Both types fall under the SIMD (Single Instruction Multiple Data) category.
- **Memory banking** is the division of memory into multiple units to allow more simultaneous data access.

Limits of SIMD

- SIMD faces limitations if the data being processed is not vectorizable, such as:
    - Linked lists
    - Any algorithm that is not vectorizable, i.e., data elements are not independent.

Performance Considerations

- Where SIMD is applicable, there is a notable performance gain. However, performance improvements are still bound by Amdahl's Law, which applies to programs only partially parallelized; the speedup is limited by the sequential portion of the program.
- Modern processors include SIMD extensions and can switch between serial and SIMD instruction modes.

Historical Context: Intel Pentium MMX

Intel Pentium MMX operations - 90s

- Concept: one instruction operates on multiple data elements simultaneously -> SIMD!
- No VLEN register.
- Opcode determines the data type:
    - 8 -> 8-bit bytes
    - 4 -> 16-bit words
    - 2 -> 32-bit doublewords
    - 1 -> 64-bit quadwords
- STRIDE is always 1.

Deep Dive into GPUs

- GPUs are essentially SIMD engines under the hood.
- The pipeline operates like a SIMD (array processor) pipeline.
- Programming is not done via SIMD instructions but through **threads**.

What are threads in GPU terminology?

- In GPU parlance, a thread is a single sequence of programmed instructions that operates on an independent set of data.
- Logical vs. Physical Resources: A thread is not directly a piece of physical hardware. Instead, it is a logical execution unit that the GPU's hardware can manage. Threads are mapped to physical hardware resources when they are scheduled for execution.
- Execution Context: Each thread has its own execution context, which includes:
    - Registers: Each thread has access to a set of registers.
    - Local memory (if any): Threads may also have access to small amounts of local memory specific to that thread.
    - Program Counter: Each thread does not independently possess a physical program counter. Instead, all threads in a **warp** share the same instruction stream and thus follow the same program counter for their execution path.

Programming Model vs. Execution Model

- **Programming model**: How the programmer writes the code, can be:

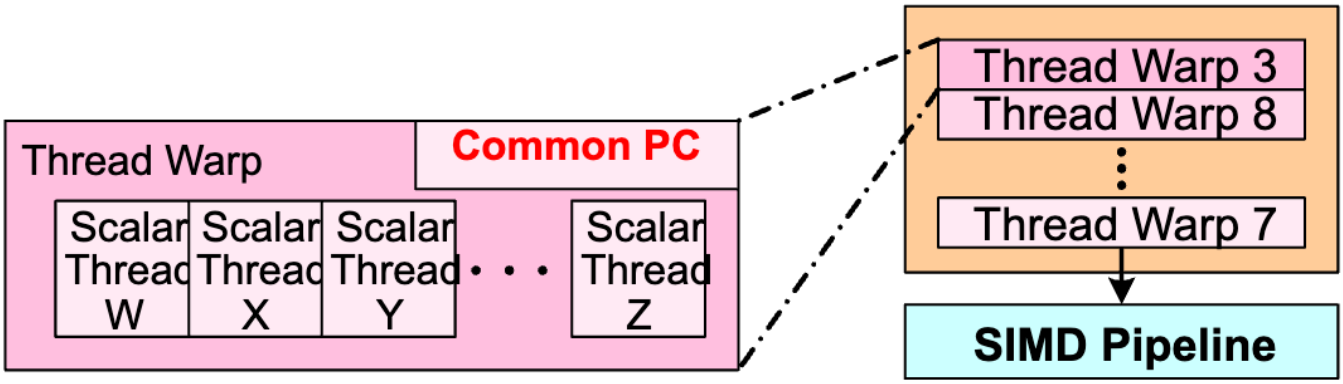
    - Sequential (von Neumann)
    - Data parallel (SIMD)
    - Dataflow
    - Multi-threaded (MIMD, SPMD)
- **Execution model**: How the hardware executes the code
    - Out-of-order execution
    - Vector processor
    - Array processor
    - Dataflow processor
    - Multiprocessor
    - Multithreaded processor
- The execution Model can be very different from the programming model
    - E.g. Von Neumann model implemented by an OoO (Out of Order) processor.
    - E.g. SPMD model implemented by a SIMD processor (a GPU)

GPUs as SIMD Machines

- In practice, a GPU is a SIMD machine without the need for SIMD instructions; it is programmed using THREADs (SPMD programming model).
    - Each thread executes the same code but on different data.
    - Each thread maintains its own context and can be used, reset, and executed independently.
- A set of threads executing the same instruction are dynamically grouped into **WARPs (wavefront)** by the hardware.
- GPUs can be viewed as SIMD machines not exposed to the programmer (SIMT = single instruction multiple threads).

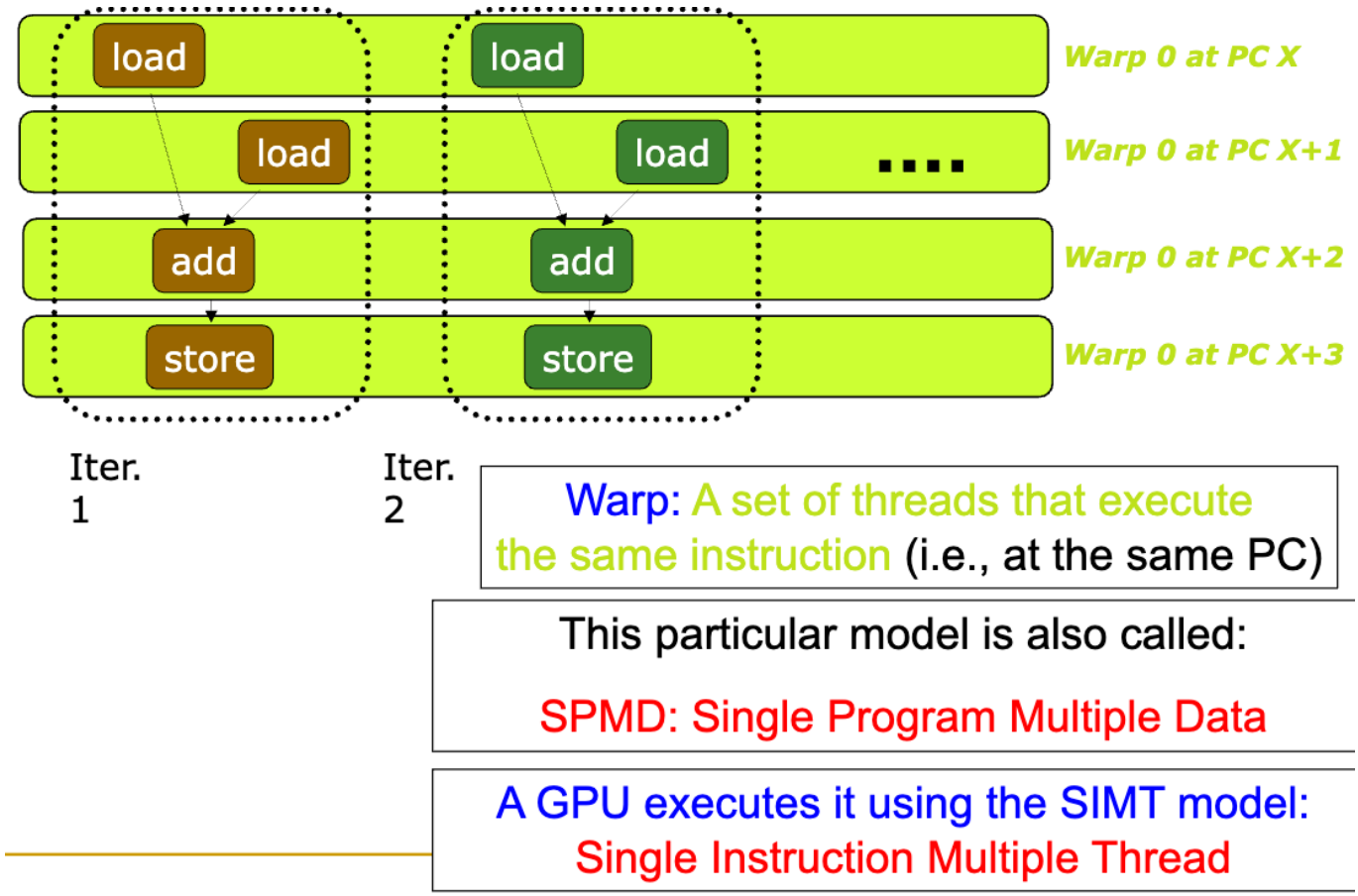
What are WARPs?

- WARPs are groups of threads that execute the same instruction. Typically, 32 threads are grouped into a WARP.
- A warp is essentially a SIMD operation formed by the hardware.
- A set of threads that execute the same instruction (on different data elements) -> SIMT (Nvidia-speak)



Example: SPMD on SIMT machine

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



How do they operate

- Uniform Instruction Execution: All threads in a warp execute the same instruction at once but on different pieces of data. This design leverages the data parallelism that GPUs are optimized for, such as in graphics rendering or scientific computations where the same operations are performed over a large set of data elements.
- Handling Divergence: If threads within a warp need to execute different instructions (due to conditional branching, for instance), this causes what's known as "warp divergence." The GPU handles divergence by serially executing each branch path needed by any thread in the warp, while other threads wait (idle). This can lead to inefficiencies and is one of the challenges in optimizing GPU code.

Warp and Thread Management:

- **Shared Execution**: Because all threads in a warp execute the same instruction simultaneously, they can be thought of as sharing a single program counter at the level of the warp. This means if threads need to execute different instructions because of conditional branching, the warp must serialize these branches, handling each outcome path one at a time, which can lead to inefficiencies.
- **Resource Allocation**: The CUDA cores (hardware execution units) do not belong to any specific thread permanently. They are allocated dynamically to threads as warps are scheduled to execute.

This means the hardware resources are shared and managed across potentially thousands of threads.

SIMD vs SIMT Execution Model

- **SIMD**: a single sequential instruction stream -> `[VLD, VLD, VADD, VST], VLEN`
- **SIMT**: multiple scalar instruction streams -> threads dynamically grouped into **warps**, effectively removing the need for a `VLEN, VMASK in SIMD -> [LD, LD, ADD, ST], NumThreads`

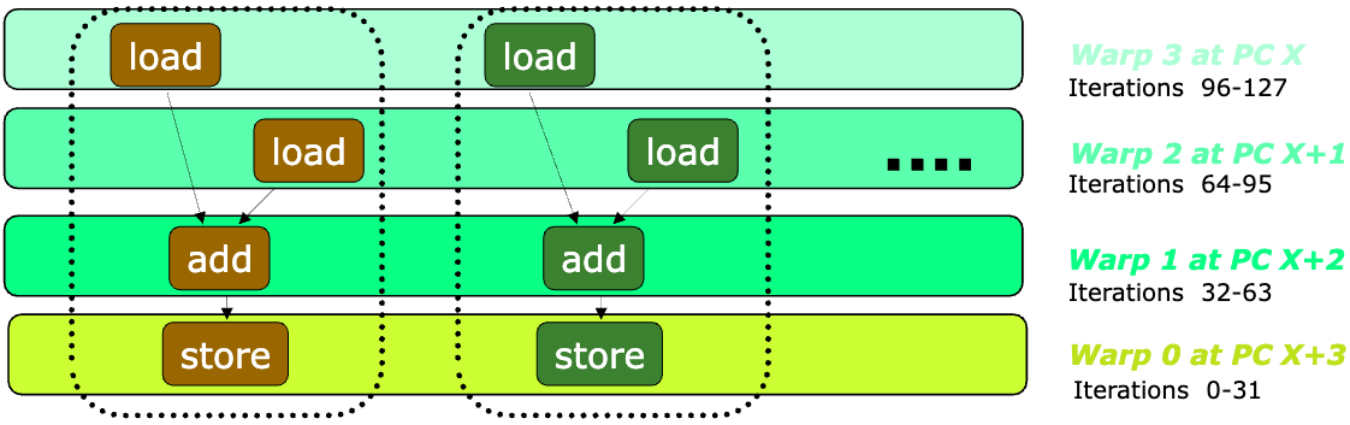
Advantages of SIMT

- In SIMT, the programmer can continue to program in the von Neumann model; the hardware takes care of the rest.
- Each thread can be treated separately -> can execute more threads independently -> MIMD processing.
- Can flexibly group threads into warps to maximize the benefits of SIMD.

Fine-grained Multi-threading of Warps

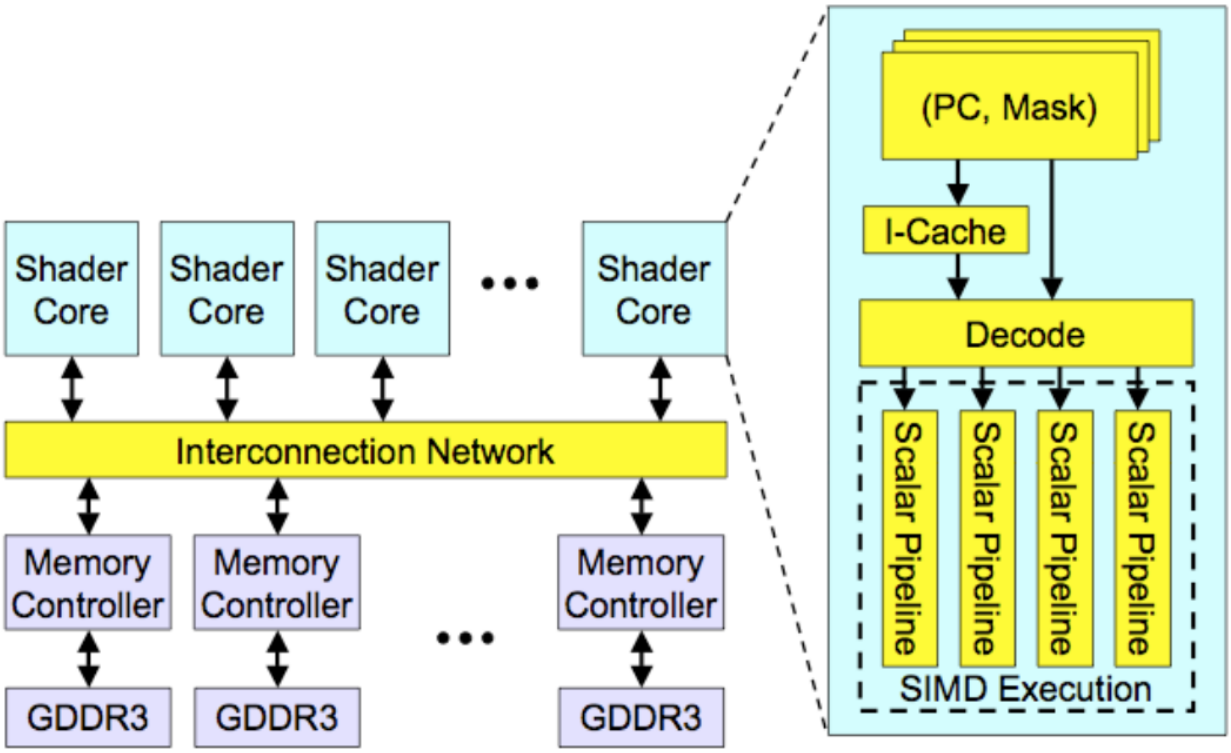
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- Warp of 32 threads.
- If there are 32k iterations & 1 iteration/thread -> 1k warps (each warp has 32 threads).
- Each warp can be interleaved on the same pipeline -> fine grained multithreading of warps



All threads in a warp are independent of each other
→ They be executed seamlessly in a fine-grained multithreaded pipeline

High-level Overview of GPU Architecture



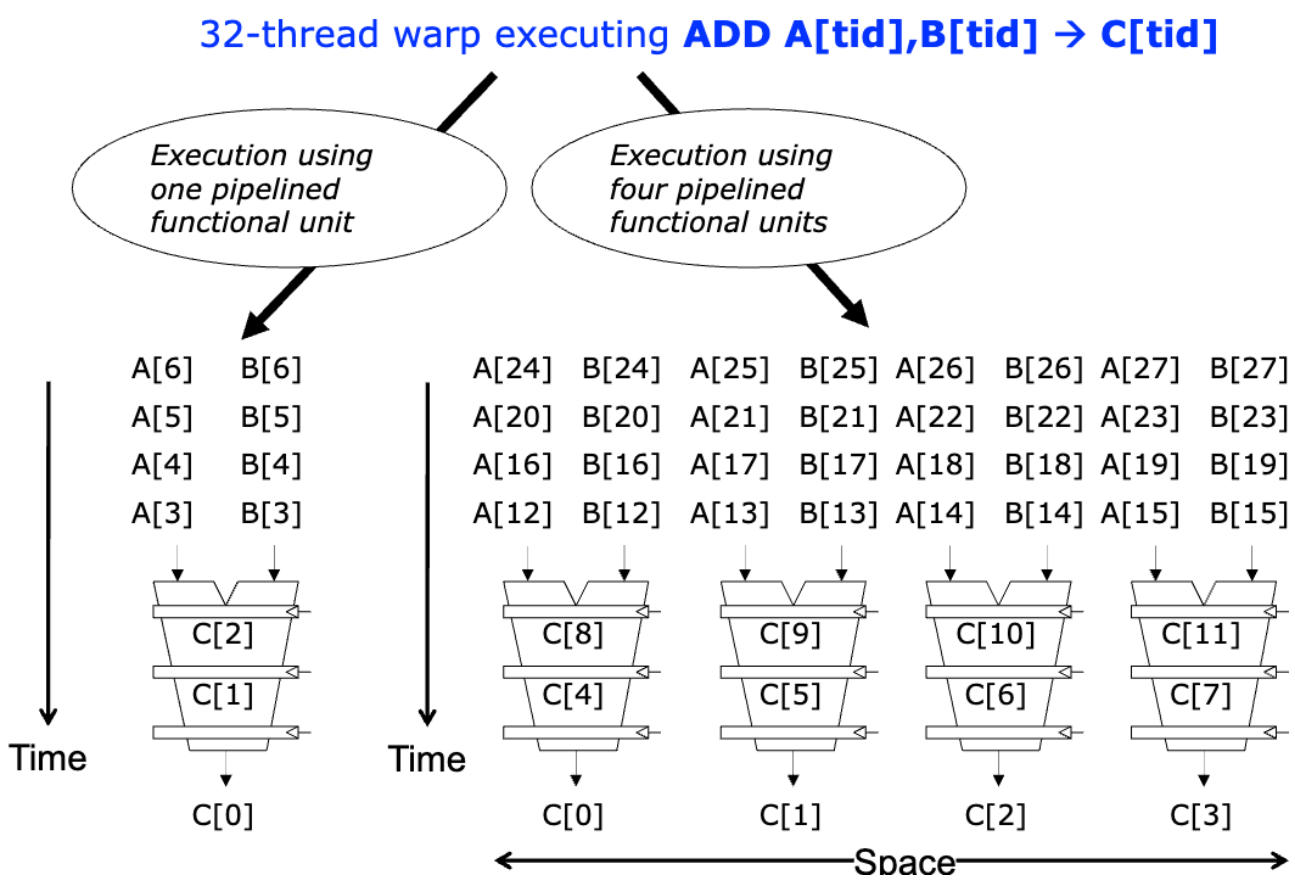
Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

GPU vs. CPU

- GPUs do not do branch prediction or check on data dependencies.
- A GPU essentially schedules warps in the pipeline.
- The pipeline remains very simple:
  - One instruction per thread at a time (no interlocking).
  - Interleaving warp execution to mask latencies.

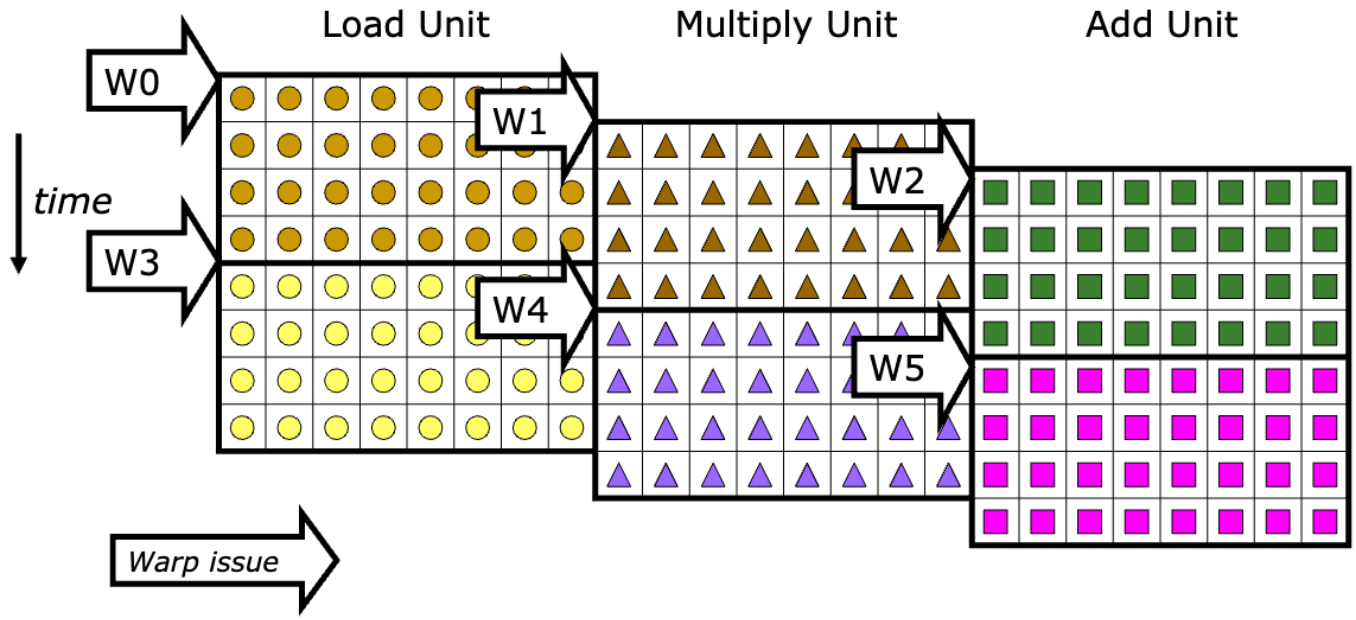
SIMD vs SIMT

- SIMD: VADD A,B -> C
- SIMT: ADD A[tid], B[tid] -> C[tid]
  - tid = threadId
  - The structure can remain the same as a SIMD processor, but using **tid**.



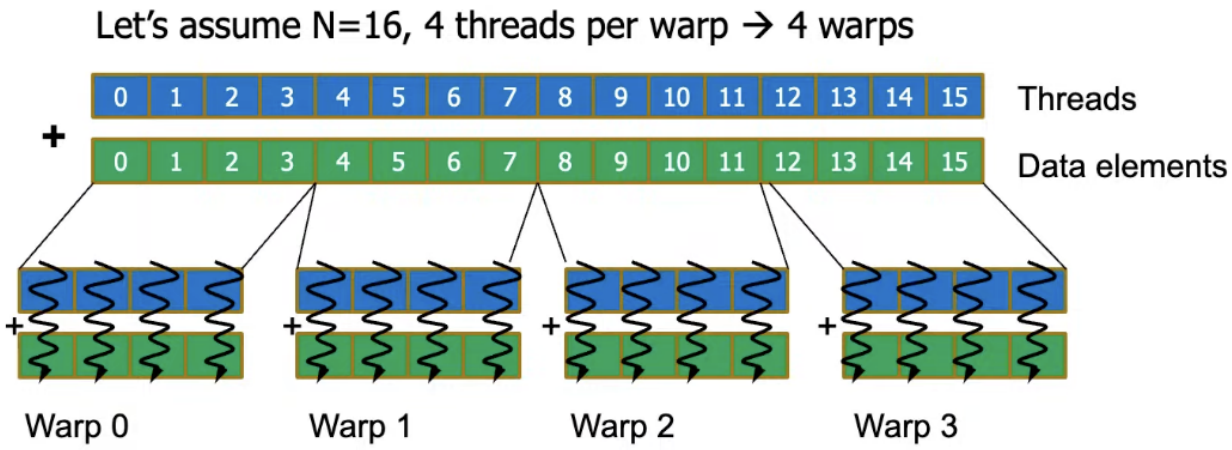
Warp Instruction Level Parallelism

- Similar to SIMD, but with threads.
- Can overlap execution of multiple instructions
  - Example machine has 32 threads per warp and 8 lanes
  - Completes 24 operations / cycle (steady state) while issuing 1 warp / cycle



Memory Access in GPU Architecture

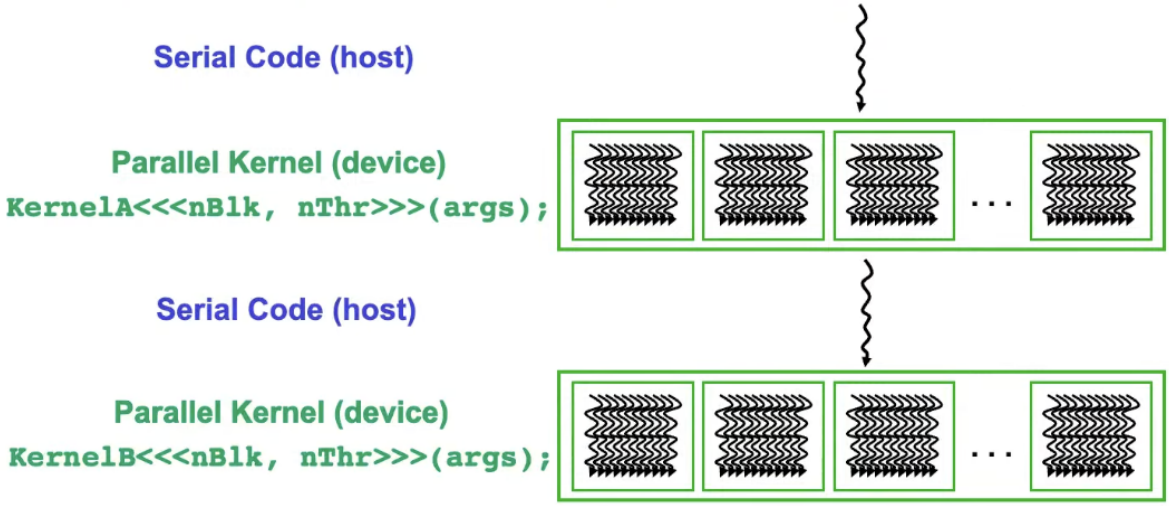
- The same instruction in different threads uses the **tid** as an index to access different data.



- When programming, it is necessary to partition the data across different threads.
- For maximum performance, the memory must have sufficient bandwidth.

Warp Exposure to Programmers

- Warps **are not exposed** to the programmer.
- **CPU threads & GPU kernels**:
  - Sequential or minimally parallelized sections on the CPU.
  - High parallelism sections on the GPU: **blocks of threads**.
  - Serial code makes sense to run on the CPU because it is better at it.



- GPUs have been very successful also because the code for a CPU is very similar to that for a GPU.

```
// CPU CODE
for (ii = 0; ii < 100000; ++ii) {
    C[ii] = A[ii] + B[ii];
}

// CUDA CODE
// there are 100000 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
```

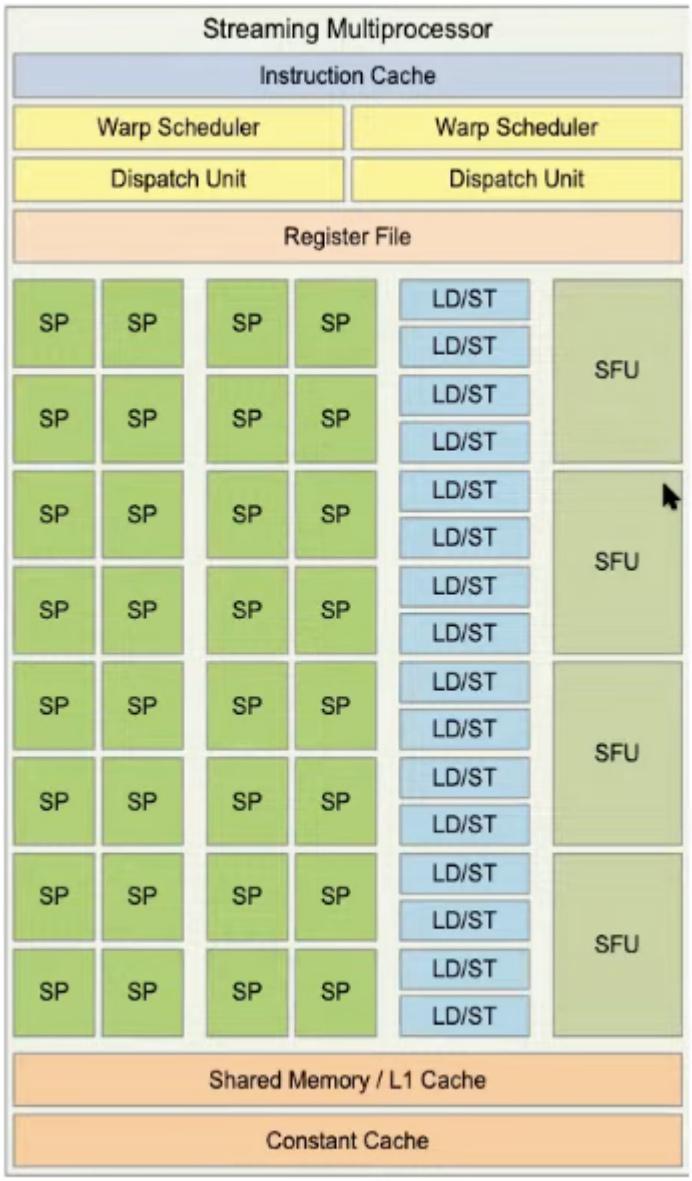
```
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

```
// CPU Program
void add matrix(float *a, float* b, float *c, int N) {
  int index;
  for (int 1 = 0; 1 < N; ++1)
  for (int j = 0; j < N; ++j) {
    index = i + j*N;
    c[index] = a[index] + b[index];
    }
  }

int main () {
  add matrix (a, b, c, N);
}

// GPU Program
__global__ add_matrix (float *a, float *b, float *c, int N){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j * N;
  if (i < N && j < N)
    c[index] = a[index] + b[index];
}

int main () {
  dim3 dimBlock(blocksize, blocksize);
  dim3 dimGrid(N/blocksize, N/blocksize);
  add_matrix <<<dimGrid, dimBlock>>> (a, b, c, N);
}
```

From Blocks to Warps in GPUs

- GPU core = a SIMD pipeline.
  - Streaming processor (SP)
  - Many such SIMD processors

- Streaming multiprocessor (SM)



NVIDIA Fermi architecture

- Blocks are divided into WARPs.
  - SIMD / SIMT unit (32 threads) Warp Configuration

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD is executed on a single thread.
  - Sequential instruction execution -> lock-step (all execute the same operation at the same time and in the same operational cycle) operations in a SIMD instruction.
  - The programming model must be SIMD (no extra threads) -> the software must know the vector length.
  - ISA contains vector / SIMD instructions.
- WARP based SIMD, more scalar threads executing in a SIMD-like manner (the same instruction executed by all threads).
  - No lock step.

- Each thread can be treated individually (different warps) -> the programming model IS NOT SIMD.
  - The software does not need to know the VLEN.
  - Multithreading and dynamic grouping of threads are possible.
- Scalar ISA. => Essentially an SPMD programming model implemented on SIMD hardware.

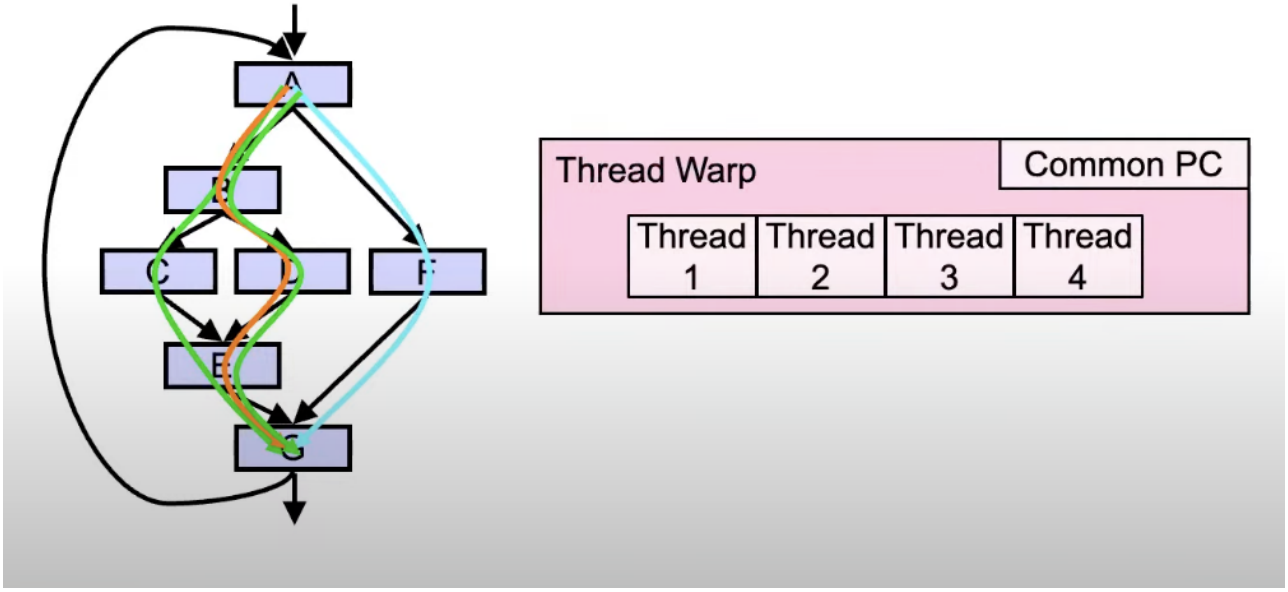
SPMD: Single Program (/ Procedure) Multiple Data

- It is a programming model, not an architectural structure.
- Each functional unit executes the same procedure but on different data.
  - The procedures can be synchronized at certain points in the program (e.g., barriers).
- Multiple execution streams run the same program.
  - Each program/procedure:
    - Works on different data.
    - Can execute a different control-flow path at runtime (!).

Flexibility of SIMT in Grouping Threads into Warps

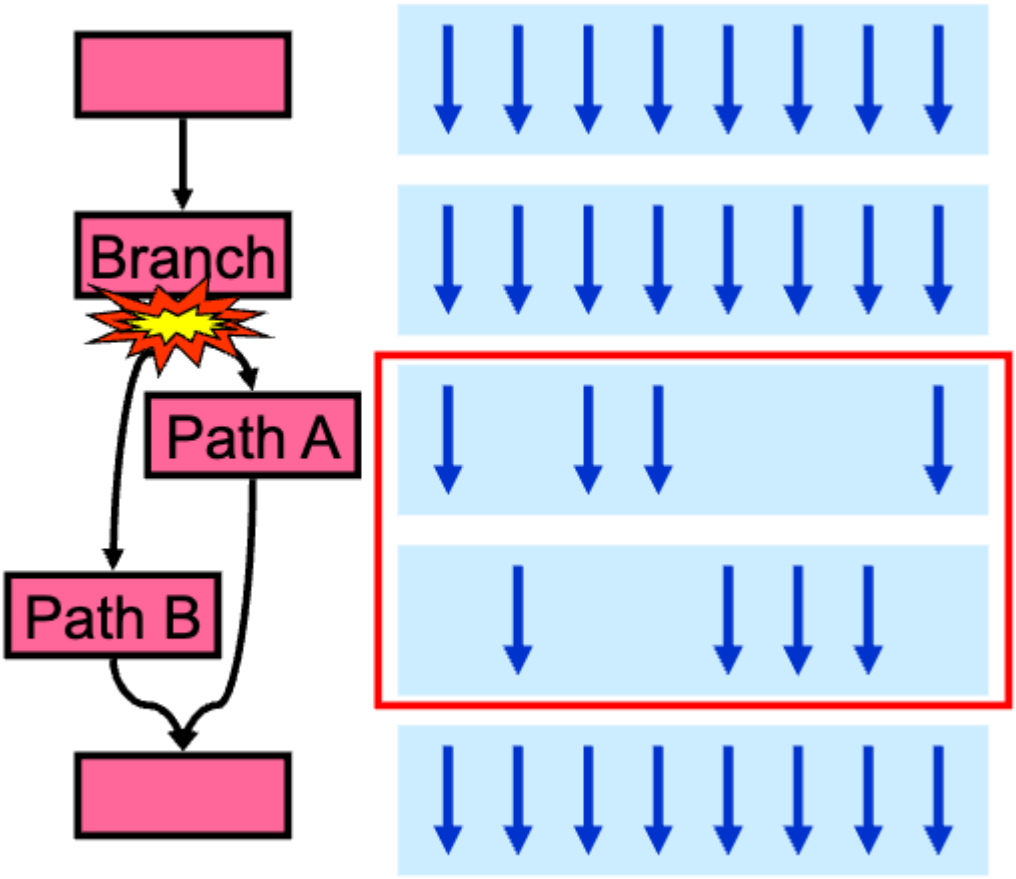
Conditional control flow instructions

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths



Control flow problem in GPUs / SIMT

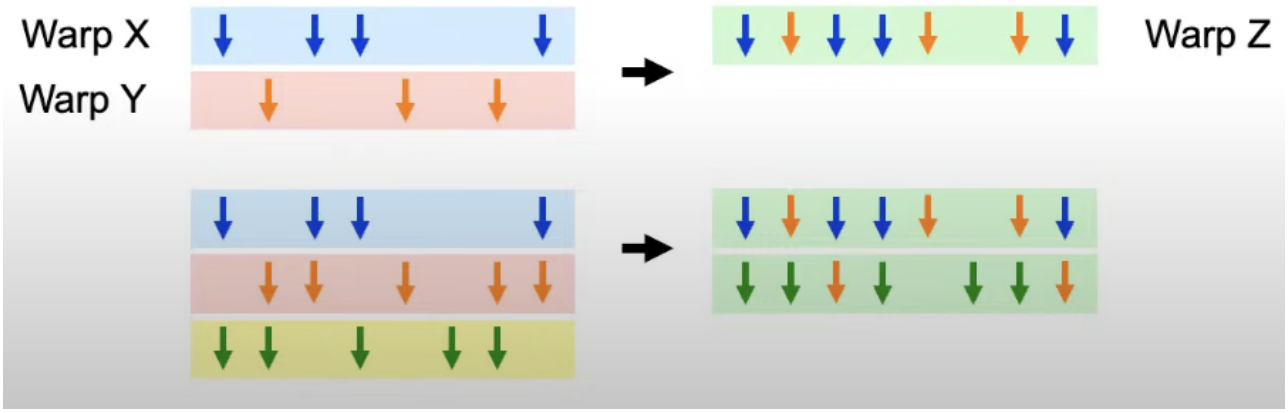
- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps
- Branch divergence occurs then threads inside warps branch to different execution paths



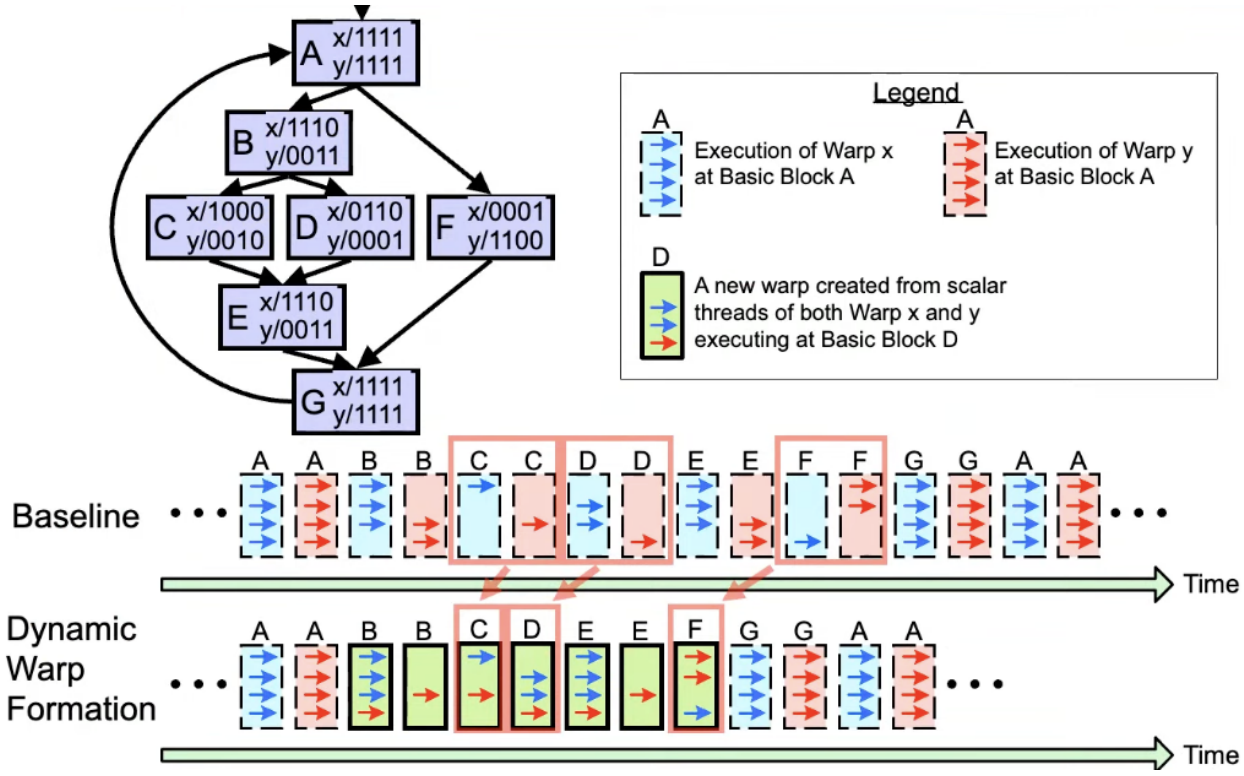
- If there are many threads, they can be:
  - Grouped more threads that are at the same PC.
  - Grouped into a single warp dynamically.
  - The result is a reduction in "divergence" -> SIMD utilization increases.
    - SIMD utilization: the fraction of SIMD lanes executing a useful operation (e.g., executing an active thread).

Dynamic Warp Formation / Merging

- The idea is to merge threads that are executing the same instruction (i.e., at the same PC) after executing a branch.
- Essentially, create new warps with waiting warps to improve SIMD utilization.



- In a complex example...



- Can threads be moved to different lines of the pipeline? -> NO

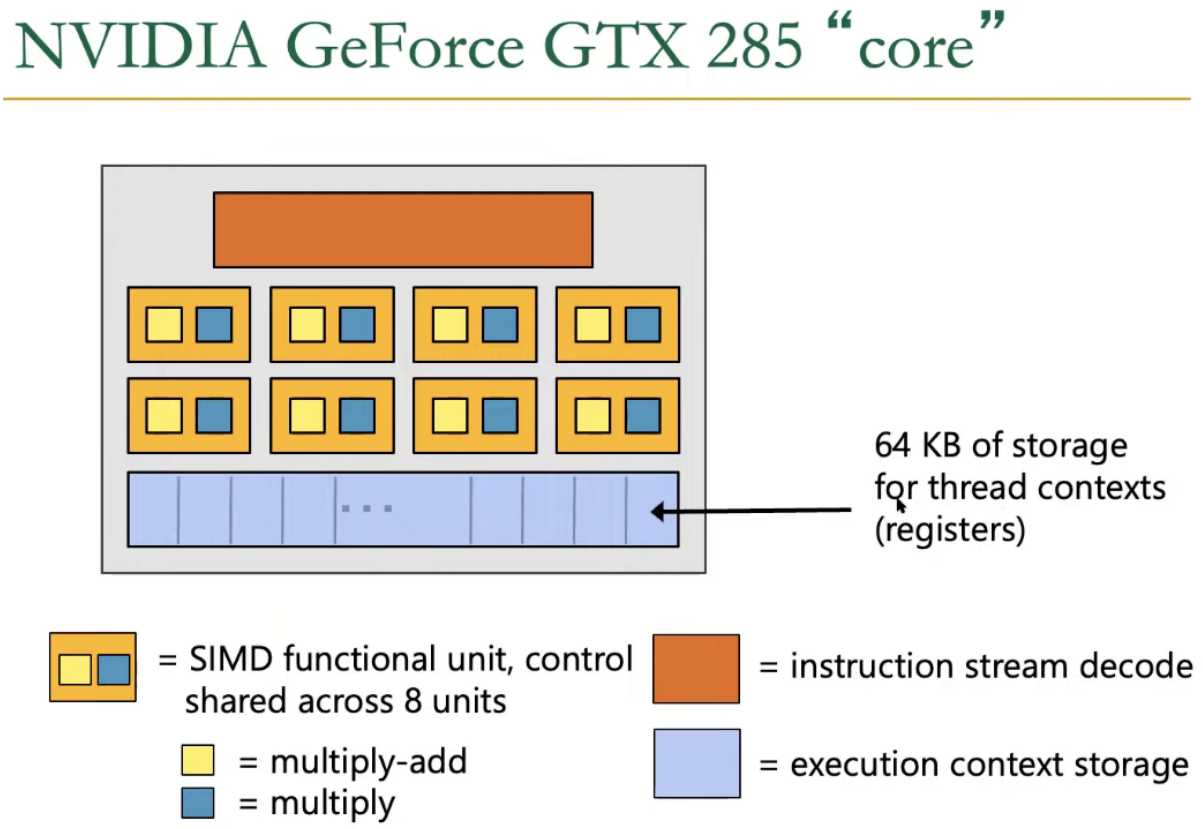
When are GPUs not efficient?

- Branch divergence: when threads in a warp take different paths.
- Long latency operations: when threads in a warp are waiting for a memory access.

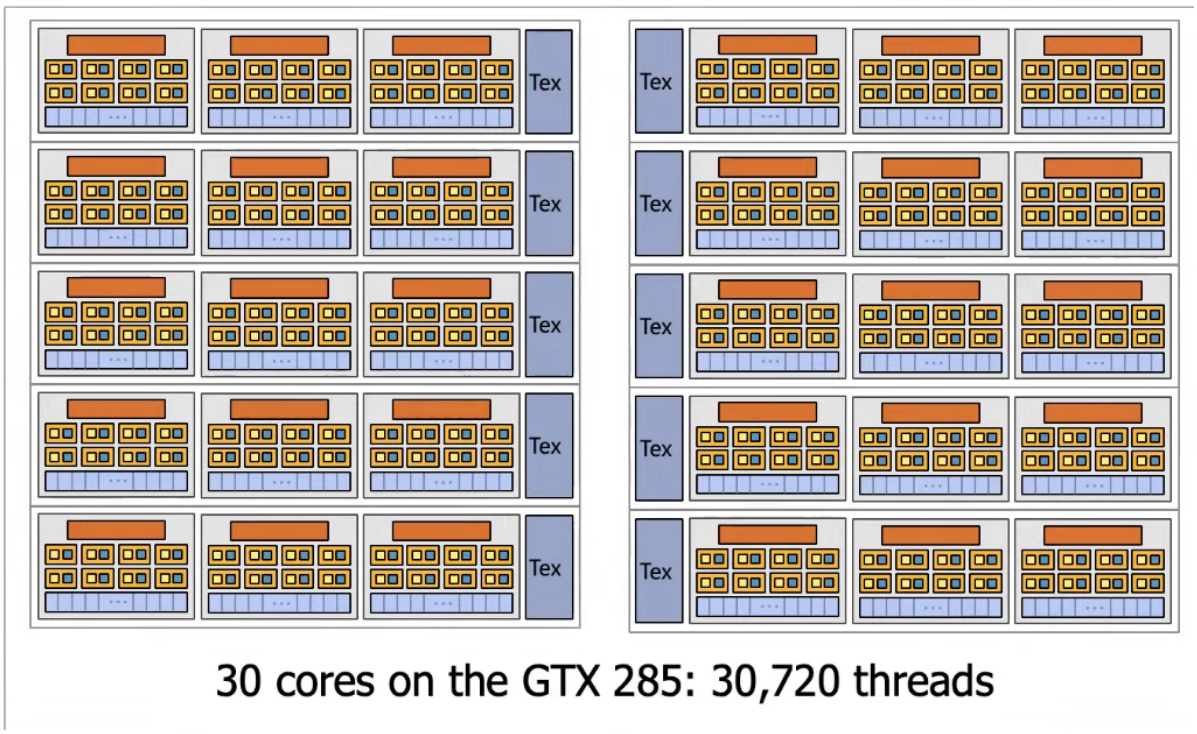
Example of a GPU - NVIDIA GeForce GTX 285

- 240 stream processors.
- SIMT execution.
- 30 cores.
- 8 SIMD FU per core.

- By today's standards, it is SMALL.



- 32 threads in a warp & 32 warps -> 1024 threads that can be used thus requiring 64 KB of storage for the threads (registers).



Tensor Cores

- What are TENSOR CORES -> essentially, they are cores specialized for performing matrix operations in an optimized manner. They are specialized cores.
- Even in tensor cores, there are SIMD processors.

SIMD vs SIMT Summary

SIMD (Single Instruction, Multiple Data) and SIMT (Single Instruction, Multiple Threads) architectures both exploit data parallelism but in different ways, leading to distinct limitations and advantages in various computing scenarios.

Limits of SIMD Architecture

See here

Limits of SIMT Architecture

1. **Thread Divergence:** In SIMT architectures like those in GPUs, thread divergence remains a challenge. When threads within the same warp choose different execution paths due to conditional branching, the GPU must execute each path serially, which diminishes the benefits of parallel execution.
2. **Resource Contention:** SIMT architectures can suffer from resource contention when multiple threads attempt to access memory or other shared resources simultaneously, leading to potential bottlenecks and reduced performance.
3. **Complex Memory Hierarchy Management:** SIMT architectures typically involve a complex memory hierarchy including local, shared, and global memory. Efficiently managing data across these memories is crucial for performance but adds to programming complexity.
4. **Synchronization Costs:** While SIMT allows each thread to execute independently, synchronization mechanisms are necessary to coordinate threads, especially when they need to share data. Implementing synchronization correctly is crucial but can be error-prone and impact performance.
5. **Software Complexity:** Although SIMT provides more flexibility than SIMD, it still requires careful programming to avoid performance pitfalls such as warp divergence and inefficient memory access patterns. The need to manage execution at the thread level can complicate software design and optimization.

Comparison and Context

While SIMD is more restrictive due to its need for uniform operations across all data elements, SIMT provides more flexibility by allowing each thread to execute independently. However, both architectures need careful management of memory access patterns and data alignment to prevent performance degradation. The choice between SIMD and SIMT typically depends on the specific requirements and constraints of the application, as well as the underlying hardware's capabilities.

21_GPU_PROGRAMMING

> https://www.youtube.com/watch?v=xz9DO-4Pkko

Topics

- GPU as an accelerator
    - Program structure
        - Bulk synchronous programming model
    - Memory hierarchy and memory management

    - Performance considerations
        - Memory access
        - SIMD utilization
        - Atomic operations
        - Data transfers
- Collaborative computing

Tensor cores

- It's a specialized hardware unit that can perform large numbers of multiplications and additions in parallel (especially for matrix multiplication).
- They are optimized to perform operations on small matrices very quickly.
- Crucial in modern deep learning & neural network training.
- They are not general-purpose, so they can't be used for arbitrary computations.
- Found within NVIDIA's newer GPUs, tensor cores operate alongside traditional CUDA cores
- To utilize tensor cores, software must be specifically written to leverage this hardware, such as CUDA and cuDNN libraries that support operations on these cores.
- They support mixed-precision computing, which allows for the use of both single and half-precision floating points.

Terminology

Generic Term	NVIDIA Term	AMD Term	Comments
Vector length	Warp size	Wavefront size	Number of threads that run in parallel (lock-step) on a SIMD functional unit
Pipelined functional unit / Scalar pipeline	Streaming processor / CUDA core	-	Functional unit that executes instructions for one GPU thread
SIMD functional unit / SIMD pipeline	Group of N streaming processors (e.g., N=8 in GTX 285, N=16 in Fermi)	Vector ALU	SIMD functional unit that executes instructions for an entire warp
GPU core	Streaming multiprocessor	Compute unit	It contains one or more warp schedulers and one or several SIMD pipelines

Why CUDA (GPUs) is better than SIMD

- As discussed in the SIMD section, the main pain point is that SIMD is:
    - Not flexible (VLEN required)
    - Not easy to program
- CUDA allows for a dynamic allocation of resources depending on the workload, enhancing flexibility in programming diverse applications.
- GPUs instead are easier to program due to the SPMD programming model.
    - GPUs have democratized High Performance Computing (HPC).
    - Greater FLOPS/$, massively parallel chip on a commodity PC (no need of a super computer, you have a power house in your PC).
- Many workloads exhibit inherent parallelism

    - Matrices
    - Image processing
    - Deep neural networks
- NVIDIA's consistent development and support for CUDA has fostered a robust ecosystem of tools, libraries, and community resources that assist developers in optimizing their applications for GPU execution.
- More scalable than SIMD

Drawbacks

- New programming model: new and existing developers must learn to think in parallel terms and adapt traditional serial code to take full advantage of GPU capabilities.
- Algorithms need to be re-implemented and rethought
- Still has some bottlenecks
    - CPU-GPU communication (PCIe bus, NVLink)
    - DRAM memory bandwidth (GDDR5, GDDR6, HBM2)
        - Even though modern gpu's memory bandwidth is very high (300-900 GB/s), it's still a bottleneck and far from the theoretical peak of the GPU.
        - The # of cores & how powerful these cores are is higher than the memory bandwidth -> GPUs are more powerful but this ratio is worse than 30 years ago.
        - Data layout: inefficient data layouts can lead to poor memory access patterns that exacerbate bandwidth limitations.
- Performance optimizations are often hardware-specific, meaning that code optimized for one generation of GPUs may not perform as well on another, requiring ongoing maintenance and updates.

CPU vs GPU

Different philosophies:

- CPU: a few **out-of-order** cores
- GPU: Many **in-order** FGMT (Fine Grained MultiThreading)

CPU

- CPUs are designed to minimize latency for a small set of instructions at a time. This design is optimized for general-purpose computing where tasks often have complex dependency chains.
- Out-of-order execution allows CPU cores to execute instructions as soon as the operands are available, rather than adhering to the original program order. This improves the efficiency of instruction pipelines by filling in execution stalls caused by instruction dependencies.
    - Requires complex hardware to manage instruction reordering and ensure correct program execution.
    - Due to the complexity of out-of-order execution, CPUs have fewer cores (powerful cores) and are optimized for single-threaded performance rather than parallelism.

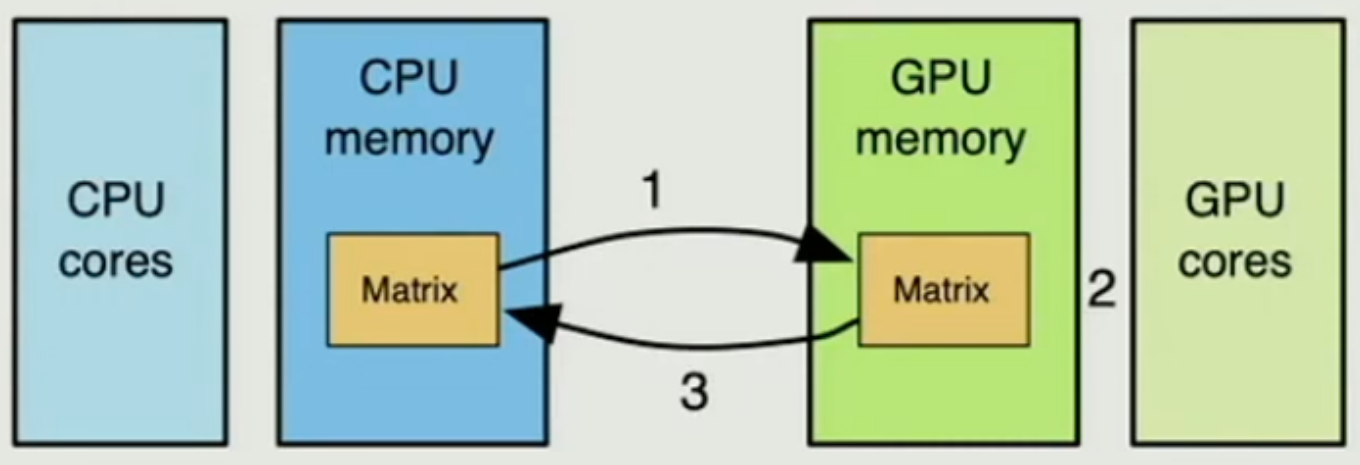
GPU

- Focus on throughput rather than latency. They are optimized for workloads that can be divided into many smaller operations that can be processed simultaneously

- GPU cores generally execute instructions in the order they are received, simplifying the hardware design and allowing for many more cores to be packed onto a single chip.
- To handle the in-order execution without suffering from latency issues, GPUs use fine-grained multithreading. This approach allows each core to switch between multiple threads in a single cycle, thus hiding execution latency caused by long-running operations or memory accesses.
- GPUs contain hundreds to thousands of simpler cores. Smaller cores are more power-efficient and can be packed more densely onto a chip, allowing for massive parallelism.

GPU computing

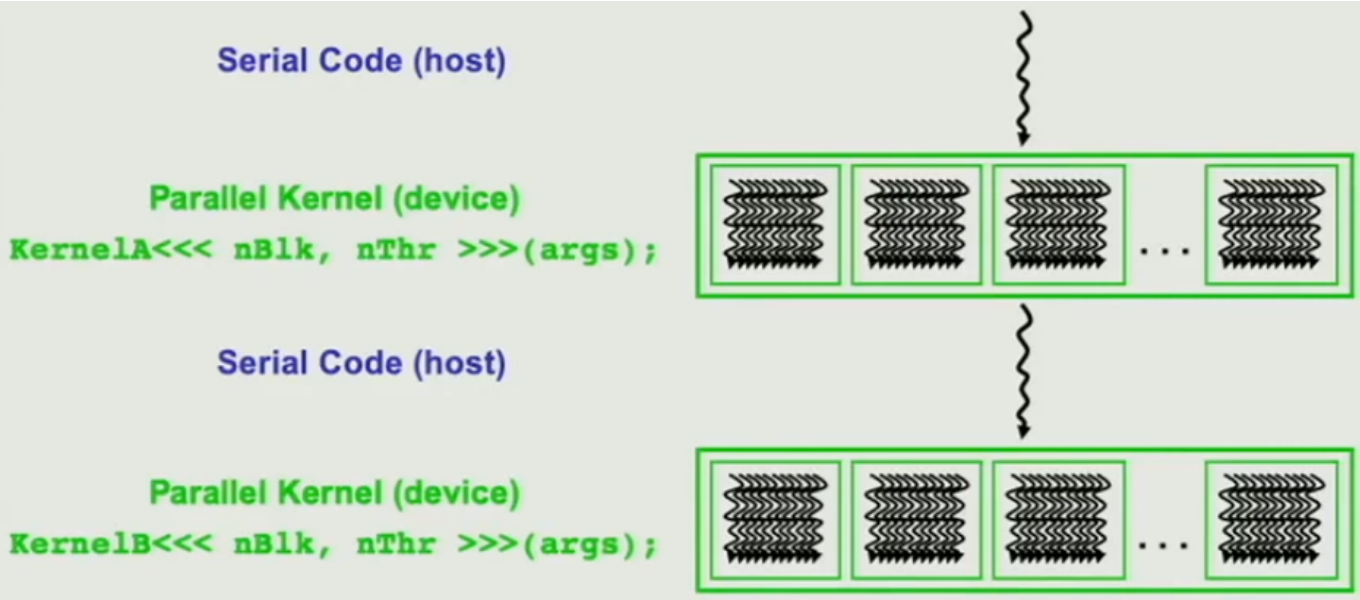
- Computation is offloaded to the GPU
- 3 steps
    1. CPU-GPU data transfer
        - This step involves latency and bandwidth considerations as data moves over PCIe or NVLink connections. The amount of data and the speed of the bus can significantly impact the overall performance of GPU-accelerated applications.
    2. GPU kernel execution
        - The performance of the GPU kernel can vary widely based on how well the kernel is optimized for the GPU's architecture.
    3. GPU-CPU data transfer



- These days it's more frequent to have SoC (System on a Chip) where the CPU and GPU are on the same chip & have access to the same memory space (e.g. Apple Silicon).
    - With shared memory, the CPU and GPU can directly access the same data without needing to copy it between separate memory pools.
    - Integrating the CPU and GPU on the same chip improves thermal management and reduces power consumption compared to systems with separate chips.
    - Still less powerful than typical discrete GPUs.
    - Well-suited for mobile devices and laptops where power efficiency is a priority.

Traditional program structure

- CPU threads and GPU kernels
- Sequential or modestly parallel code runs on the CPU
- Massive parallelism runs on the GPU



Terminology

- Host <-> CPU
- Device <-> GPU

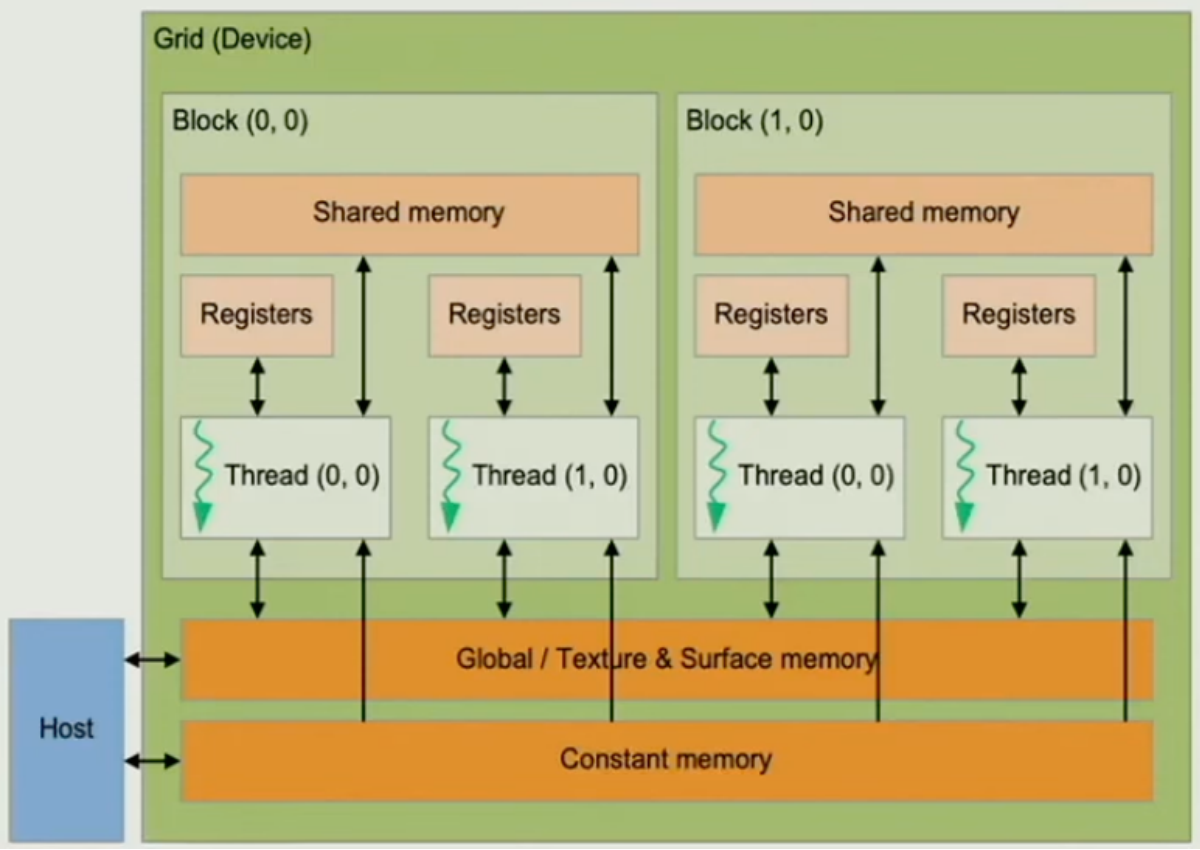
Recall: SPMD

> As seen here SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than a hardware model.
- Each processing element executes the same procedure, but on different data.
  - Procedures can synchronize and communicate (barriers)
- Multiple instructions streams execute the same program
  - Each program
    1. Works on a different part of the data
    2. Can execute different control flow, at run-time

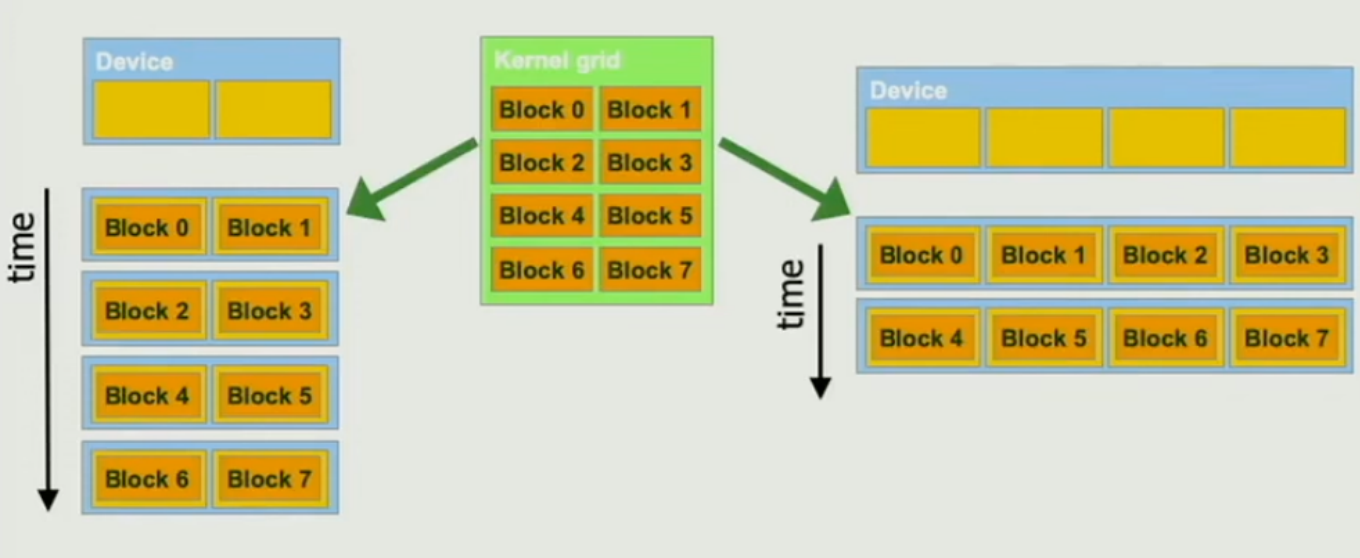
CUDA / OPENCL Programming Model

- SIMT or SPMD
- Bulk synchronous programming model
  - All threads in a block execute the same instruction at the same time
  - All threads in a block are synchronized at the end of each instruction
  - All threads in a block can communicate and share data
  - Threads in different blocks can't communicate
  - Global (coarse-grained) synchronization is possible between kernels
- The host (CPU typically) allocates memory, copies data and launches kernels
  - Grid: collection of blocks
  - Block: collection of threads
    - Within a block, shared memory and synchronization
  - Thread: single execution unit



- The only way to synchronize threads inside a kernel is by terminating the kernel (global synchronization)
  - Inside a block, threads can synchronize using barriers

- Hardware is free to schedule thread blocks and we do not have control over it (the order of execution of blocks is not guaranteed). Even though we know that the scheduling is done with round-robin.
- This is the reason why that all the blocks have finish is by terminating the kernel.



Traditional program structure in CUDA

- Function prototypes

```
float serialFunction(...);
__global__ void kernel(...);
```

- Main function

1. **Allocate memory** space on the device -> `cudaMalloc(&d_in, bytes)`
2. Transfer data from **host to device** -> `cudaMemCpy(d_in, h_in, bytes, cudaMemcpyHostToDevice)`
3. Execution configuration setup
   - # of blocks
   - # of threads per block
4. **Kernel launch** -> `kernel<<<numBlocks, numThreadsPerBlock>>>(d_in, d_out)`
5. **Transfer data back** from device to host -> `cudaMemCpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost)`
6. Repeat as needed*

- Kernel `__global__ void kernel(type args, ...)`
  - To distinguish from host functions
  - Automatic variables transparently assigned to registers
  - Shared memory: `__shared__`
  - Intra-block synchronization: `__syncthreads()`

- Memory allocation

    - `cudaMalloc(&d_in, bytes)`
    - `cudaMemCpy(d_in, h_in, bytes, cudaMemcpyHostToDevice)`
    - `cudaMemCpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost)`
    - `cudaFree(d_in)`

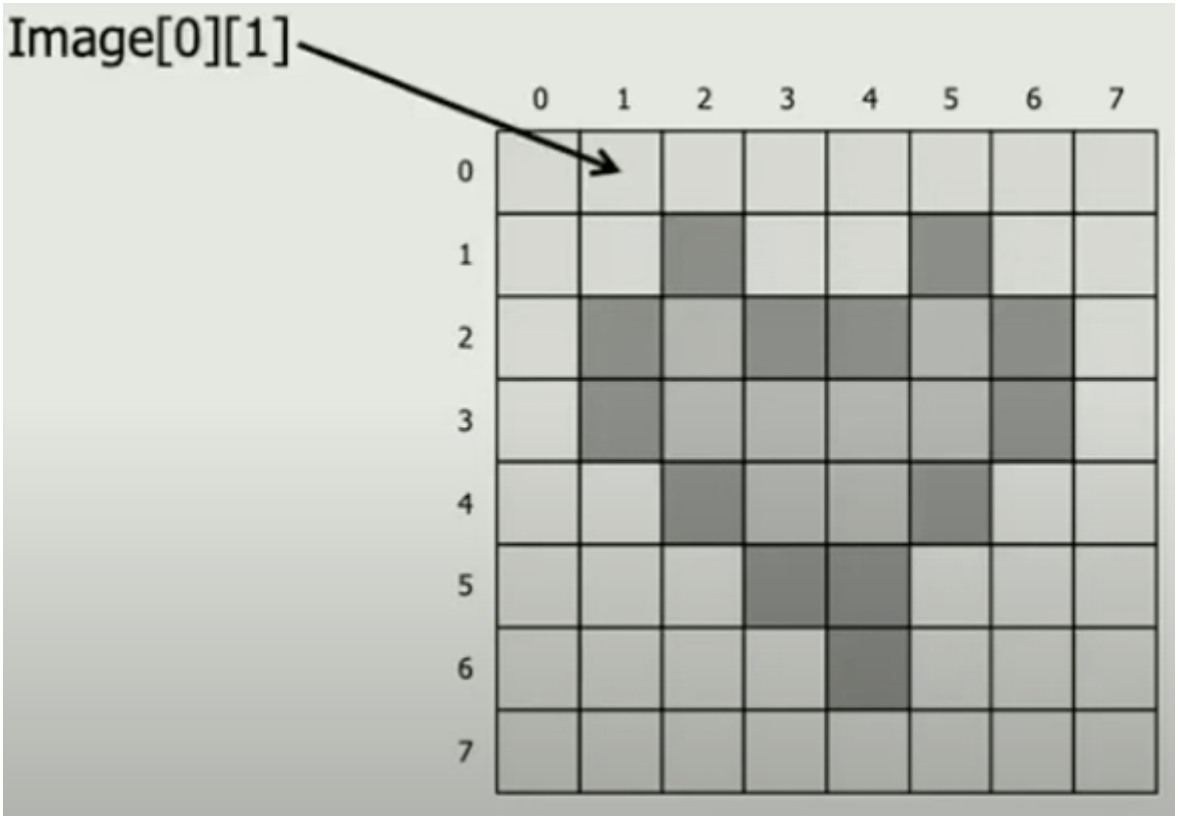
- Kernel launch
    - `kernel<<<numBlocks, numThreadsPerBlock>>>(d_in, d_out)`

- Memory deallocation
    - `cudaFree(d_in)`

- Explicit synchronization
    - `cudaDeviceSynchronize()`
    - Where do we use it? We use it in the host code right after the kernel call to make sure that the kernel has finished executing. Why? It's the only way to know that the kernel has finished executing. The kernel call is asynchronous, so the host code will continue executing after the kernel call even though the kernel is still executing.

Indexing and memory access

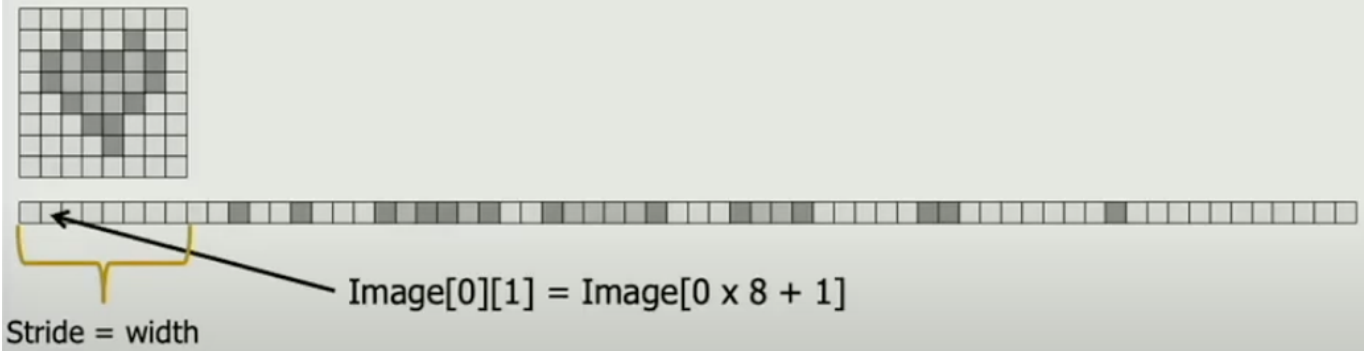
- Images are 2D data structures
    - height x width
    - `Image[j][i], where 0<=j<height, 0<=i<width`



How is it actually stored in memory?

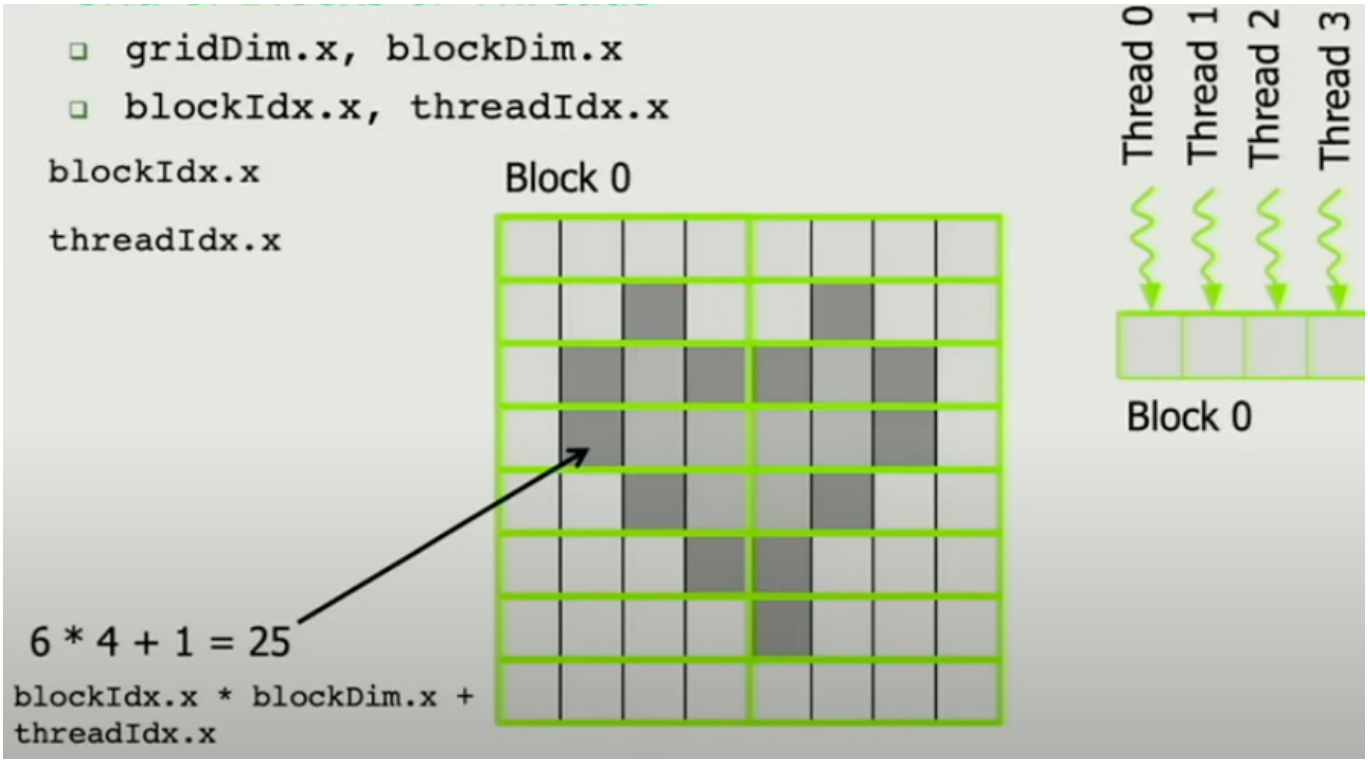
- **Row-major layout**

- `Image[j][i]` is stored in memory as `Image[j*width + i]`



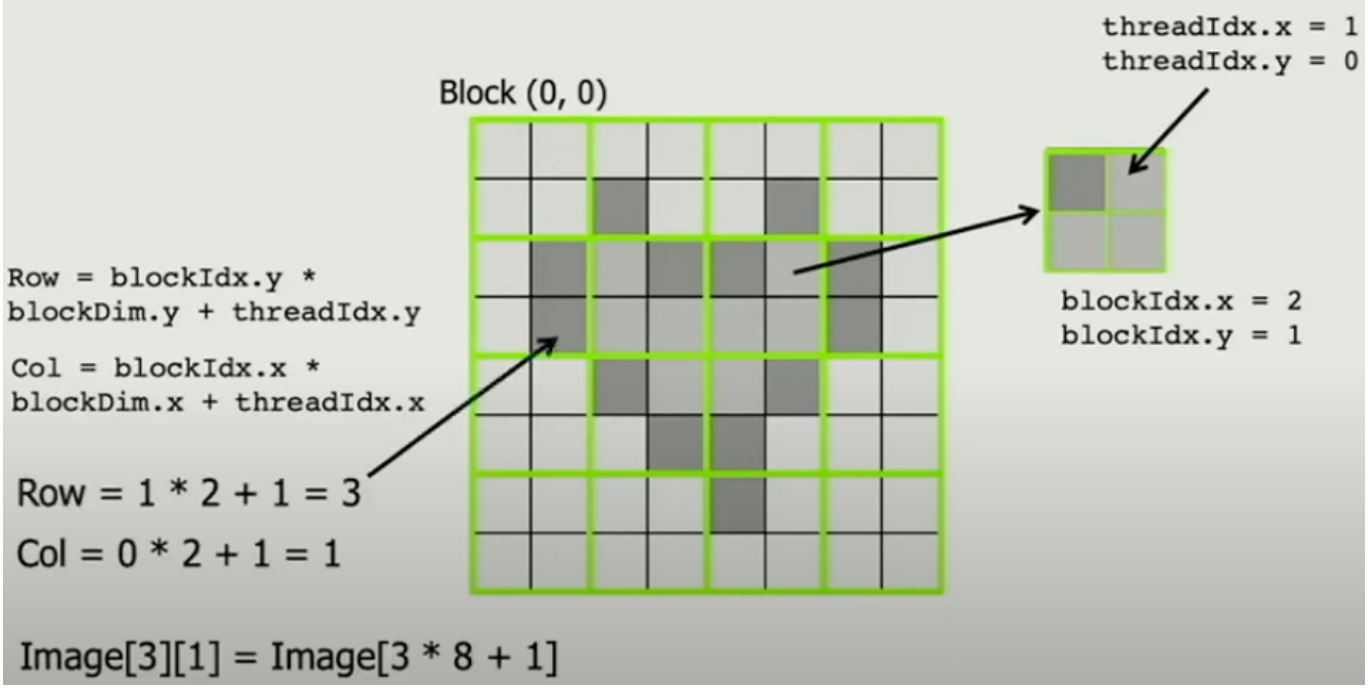
1D Grid (One way to do it)

- One GPU thread per pixel
- Grid of blocks of threads
  - `gridDim.x` -> # of thread blocks that the grid contains in the x direction
  - `blockDim.x` -> # of threads that the block contains in the x direction
  - `blockIdx.x` -> index of the block in the grid in the x direction
  - `threadIdx.x` -> index of the thread in the block in the x direction



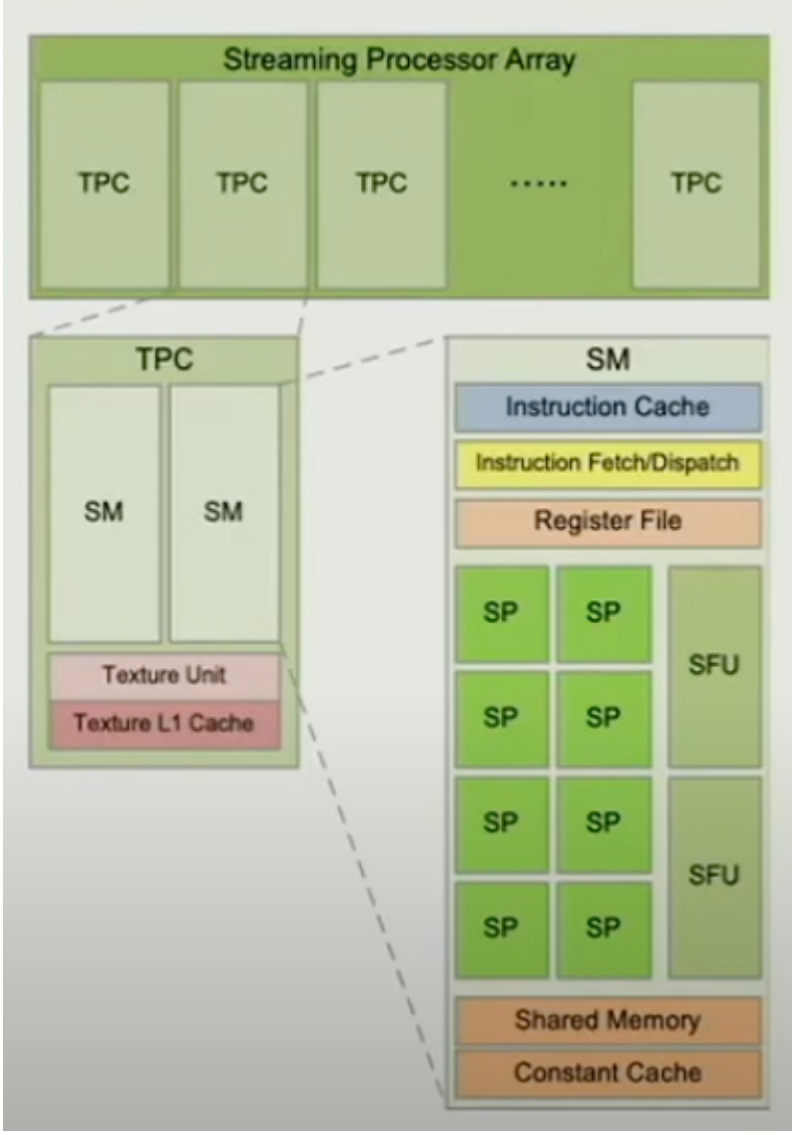
2D Grid (Another way to do it)

- `gridDim.x` -> # of thread blocks that the grid contains in the x direction
- `gridDim.y` -> # of thread blocks that the grid contains in the y direction

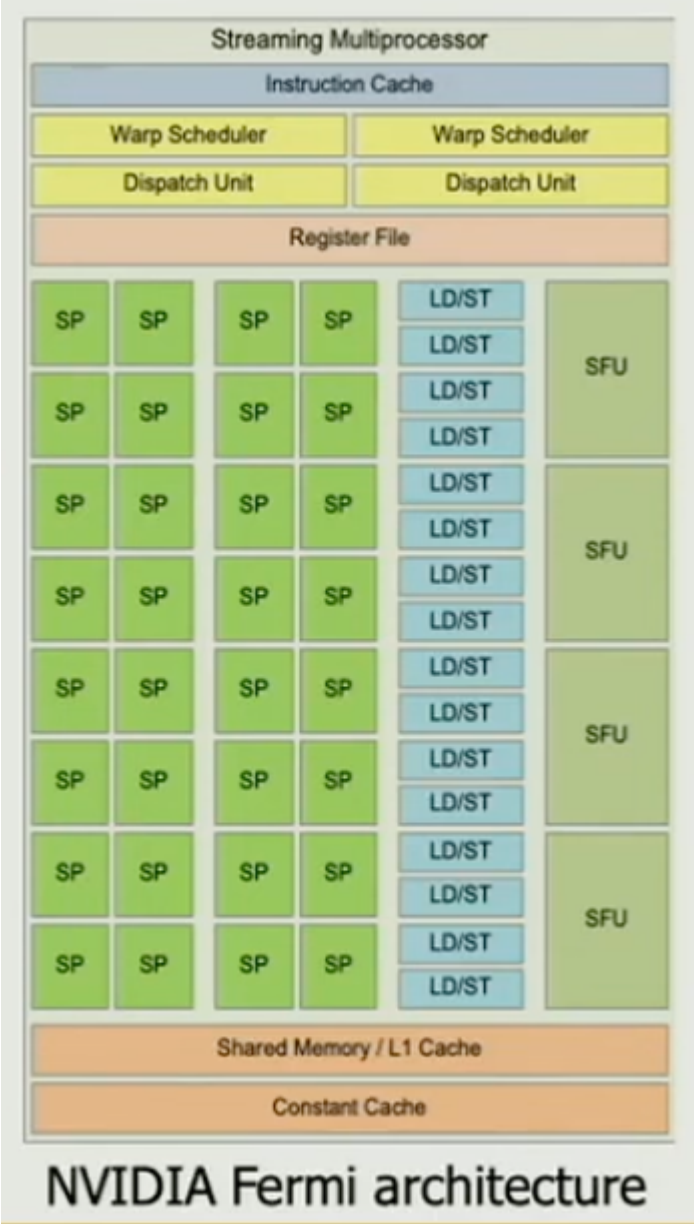


Review of GPU Architecture

- Introduced in 2007, the Tesla architecture marked a significant step towards GPGPU (General-Purpose computing on Graphics Processing Units). It includes an array of Streaming Processors (SPs) which are essentially small, efficient cores designed to handle multiple threads simultaneously.



- Streaming Multiprocessor (SM)
    - Each SM contains several SPs (Streaming Processors) that perform the actual computations. SPs are capable of executing integer and floating-point operations.
- Blocks are divided into **warps**
    - SIMD unit (32 threads)
    - Uses scoreboarding (internally) to manage the execution of threads



- Streaming Multiprocessors (SM) or Compute units (CU): essentially clusters of SIMD pipelines which are organized to optimize parallel processing of data.
    - SIMD pipelines
- Streaming Processors (SP) or CUDA cores: often referred to as vector lanes, these cores are where the computations are carried out. Each core can handle one thread at a time.
- Number of **SMs x SPs** across generations
    - **Tesla (2007)**: 30 SMs x 8 SPs per SM
    - **Fermi (2010)**: 16 SMs x 32 SPs per SM
    - **Kepler (2012)**: 15 SMs x 192 SPs per SM
    - **Maxwell (2014)**: 24 SMs x 128 SPs per SM
    - **Pascal (2016)**: 56 SMs x 64 SPs per SM
    - **Volta (2017)**: 84 SMs x 64 SPs per SM
    - **Turing (2018)**: 68 SMs x 64 SPs per SM
    - **Ampere (2020)**: 108 SMs x 64 SPs per SM
    - **Hopper (2022)**: 144 SMs x 128 SPs per SM

    - **Lovelace (2022)**: 144 SMs x 128 SPs per SM

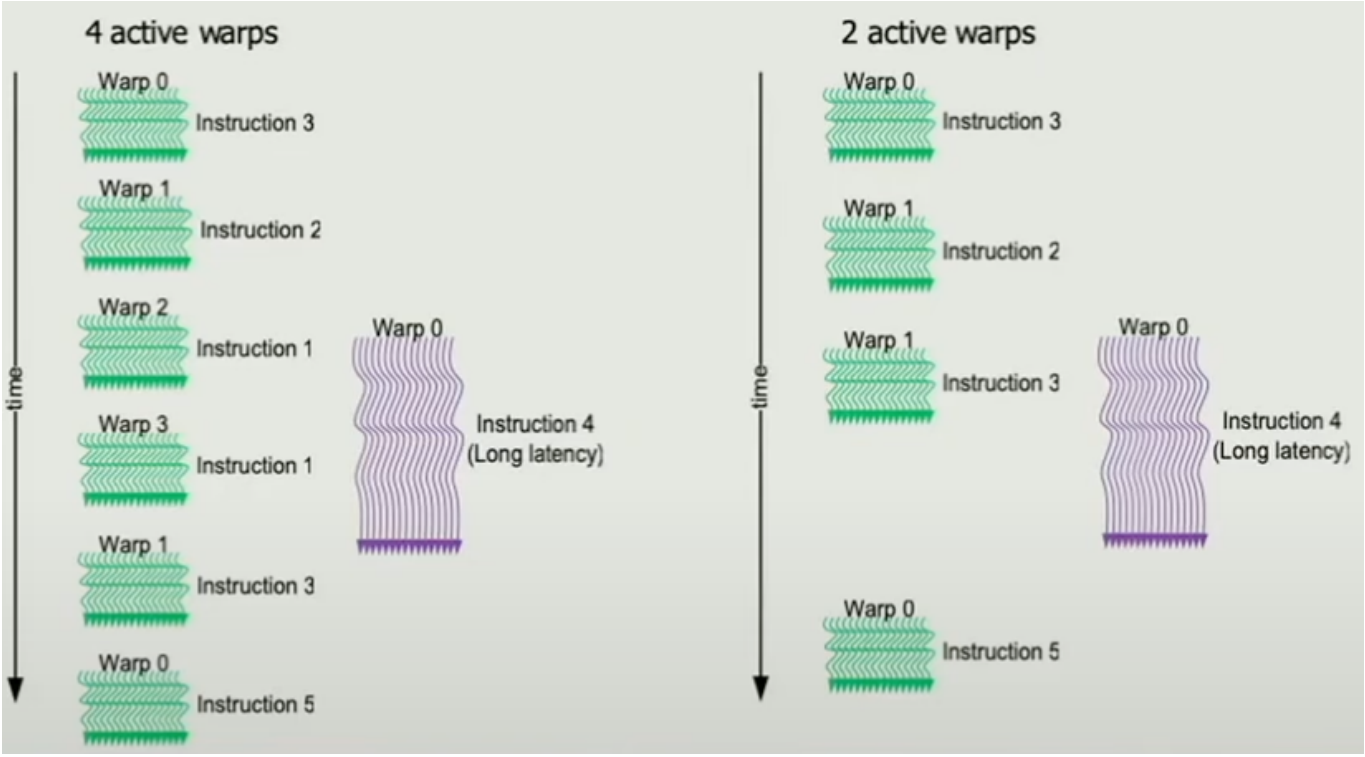
Performance considerations

- **Main bottlenecks**
    - **Global memory access** (the main GPU memory) is often a significant bottleneck due to its relatively high latency and limited bandwidth compared to the compute capabilities of the GPU cores.
    - **CPU-GPU data transfers** over the PCIe bus can substantially affect performance, particularly for applications that require frequent data exchanges.
- **Memory access**
    - **Latency hiding**
        - Occupancy refers to the number of warps that are active on a multiprocessor at a given time. Higher occupancy can help hide the latency of memory access because while some threads are waiting for data to arrive from memory, others can be executing
    - **Memory coalescing** refers to the optimization technique where adjacent threads access consecutive memory addresses. When memory access is coalesced, multiple data elements can be loaded in a single memory transaction, which reduces the number of required memory accesses and improves bandwidth utilization.
    - **Data reuse**
        - **Shared memory usage** can significantly reduce the reliance on slower global memory. Shared memory is much faster but limited in size
- SIMD (Warp) utilization
    - **Branch divergence** occurs when threads of the same warp follow different execution paths due to conditional statements. This divergence can lead to serialization within the warp, where some threads are idle while others are executing, reducing overall efficiency.
    - **Memory access patterns**, crucial for maximizing throughput.
        - Strided accesses, where threads access memory locations that are spaced apart, can lead to poor utilization of memory bandwidth.
- **Atomic operations** ensure that a particular memory location is updated atomically, preventing race conditions between threads.
    - **Avoid them if possible**
    - **Use them sparingly**
    - **Serialization**: two threads trying to write to the same memory location. One must wait for the other to complete, which can serialize the operations and degrade performance.
        - Overlap of communication and computation to mitigate the impact of data transfer times.
            - Asynchronous data transfers can help overlap communication and computation, allowing the CPU to perform other tasks while the GPU is processing data.

Memory access - Latency hiding

- **FGMT** (Fine-Grained Multi-Threading) helps to hide the latency of long memory accesses by interleaving the execution of multiple threads or warps. When one warp stalls due to a memory request, another warp can take over the execution unit.
- **Occupancy**: It is defined as the ratio of active warps to the maximum number of warps that can be active per Streaming Multiprocessor (SM). Basically how many warps are active at the same time.

- **Active warps**: warps that are currently executing and not stalled waiting for resources.
- **Maximum warps**: total number of warps that an SM can support at any time.
- **Maximum number of blocks per SM**: each can typically support up to 32 blocks.
- **Register usage** & **Shared memory usage**: fixed amount of register and shared memory, often around 256KB for registers and 64KB for shared memory in modern architectures.



Example

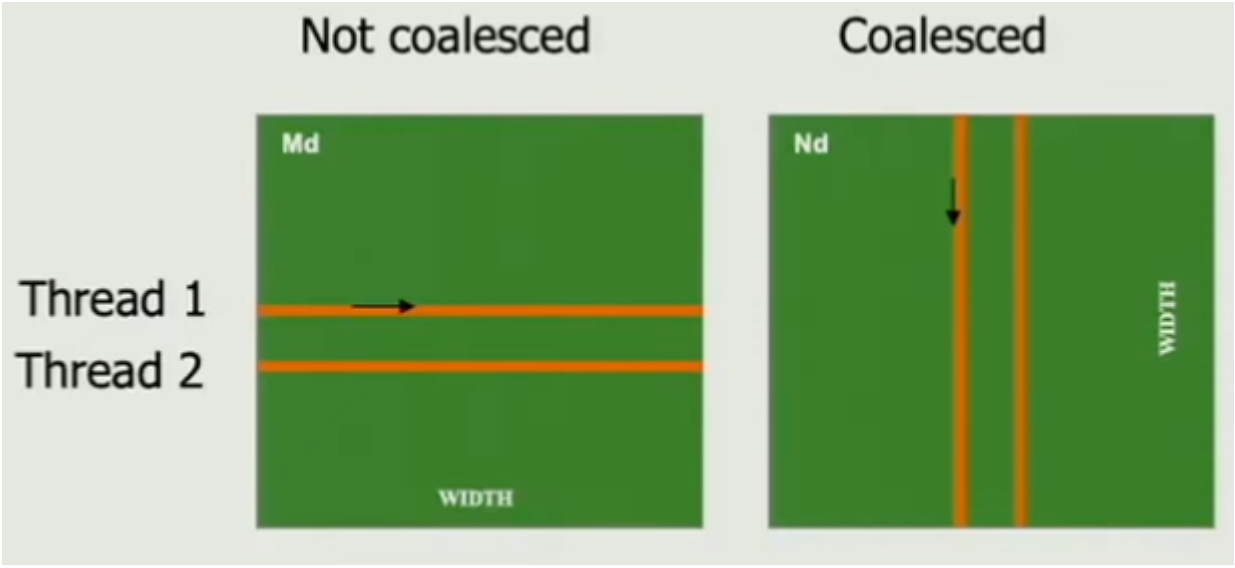
Consider a scenario where each thread block in a GPU program requires 30KB of shared memory. Given that the typical shared memory available per SM is 64KB, you can calculate the number of blocks that can fit within one SM:

Calculation: Since each block requires 30KB, only two blocks can be accommodated within one SM (2 * 30KB = 60KB < 64KB). Crucial because fitting more blocks might exceed the available shared memory, leading to a reduction in occupancy and potential performance degradation.

- Occupancy Calculation
  - **Number of Threads per Block**: Defined by the programmer, this affects how threads are distributed across warps and blocks.
  - **Number of Registers per Thread**: Known at compile time, more registers per thread reduce the number of threads (and thus warps) that can be active at any one time.
  - **Shared Memory per Block**: Also defined by the programmer, this is the amount of shared memory each block requires. High shared memory usage can limit the number of blocks per SM.

Memory access - Coalescing

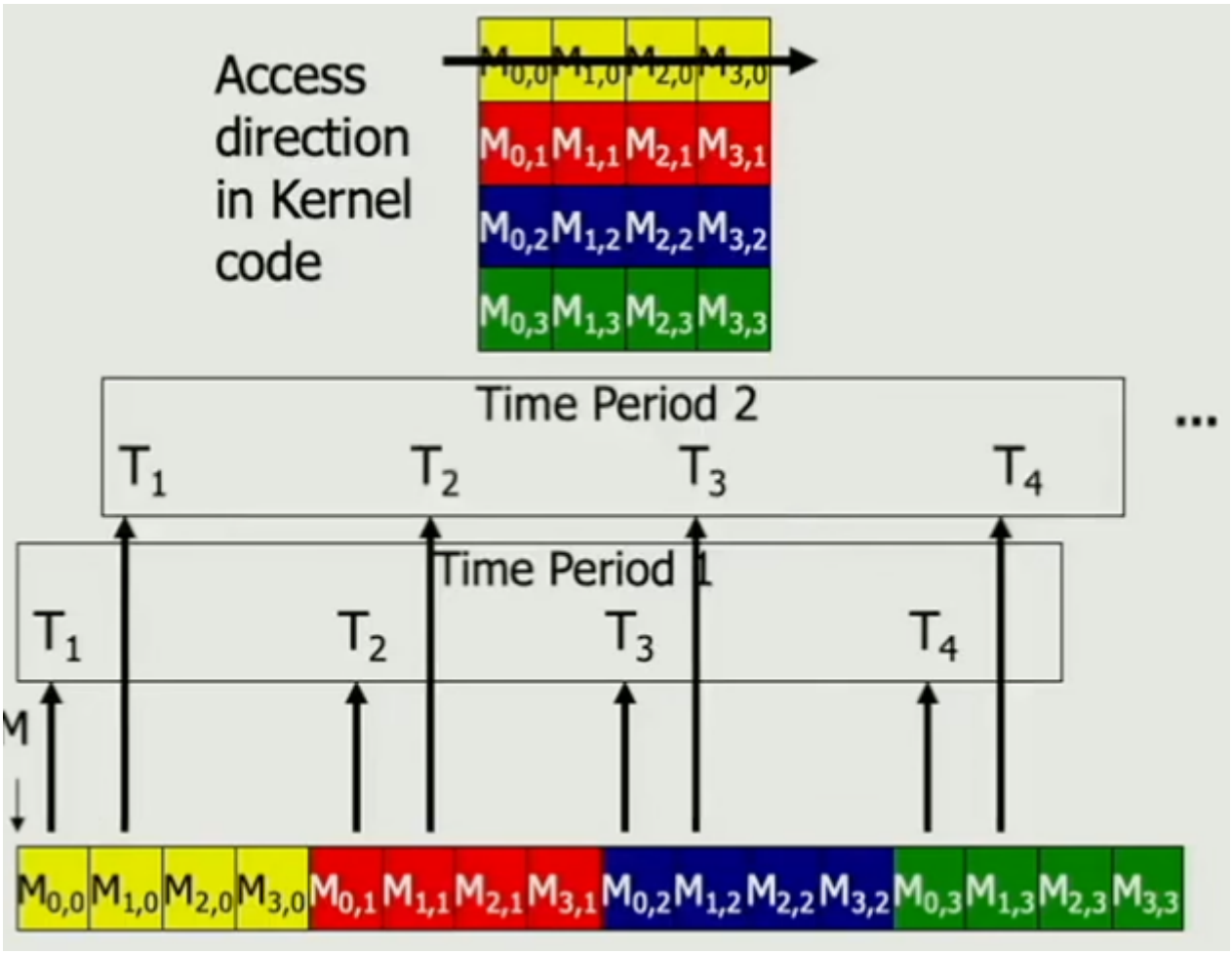
- It involves structuring memory accesses by threads within a warp to ensure that they are as efficient as possible, ideally leading to peak memory bandwidth utilization.
- **Peak bandwidth** utilization occurs when all threads in a warp access one cache line



> When threads in a warp access global memory, it is most efficient if they access consecutive memory addresses. This allows the GPU to combine these accesses into a single memory transaction instead of multiple, reducing the number of memory accesses and maximizing bandwidth utilization.

Example - Uncoalesced memory accesses

Threads access memory locations that are not adjacent, leading to multiple memory transactions. This is less efficient and increases the latency of memory operations.

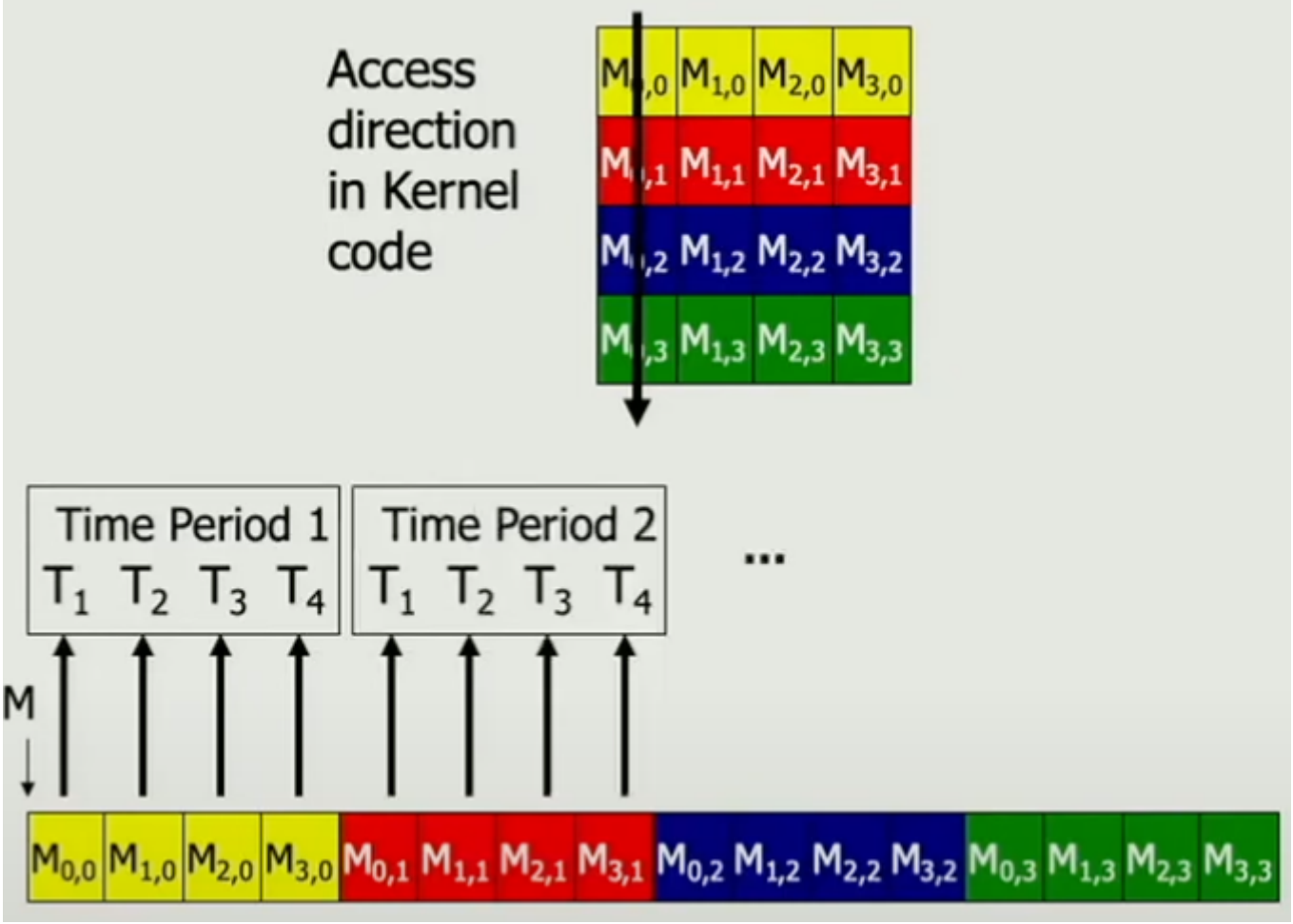


> T stands for thread
>
> Time period stands for Iteration

This causes 4 memory transactions.

Example - Coalesced memory accesses

Threads in a warp access one cache line, this results in a single memory transaction, significantly reducing the memory access time and increasing the efficiency of the warp.

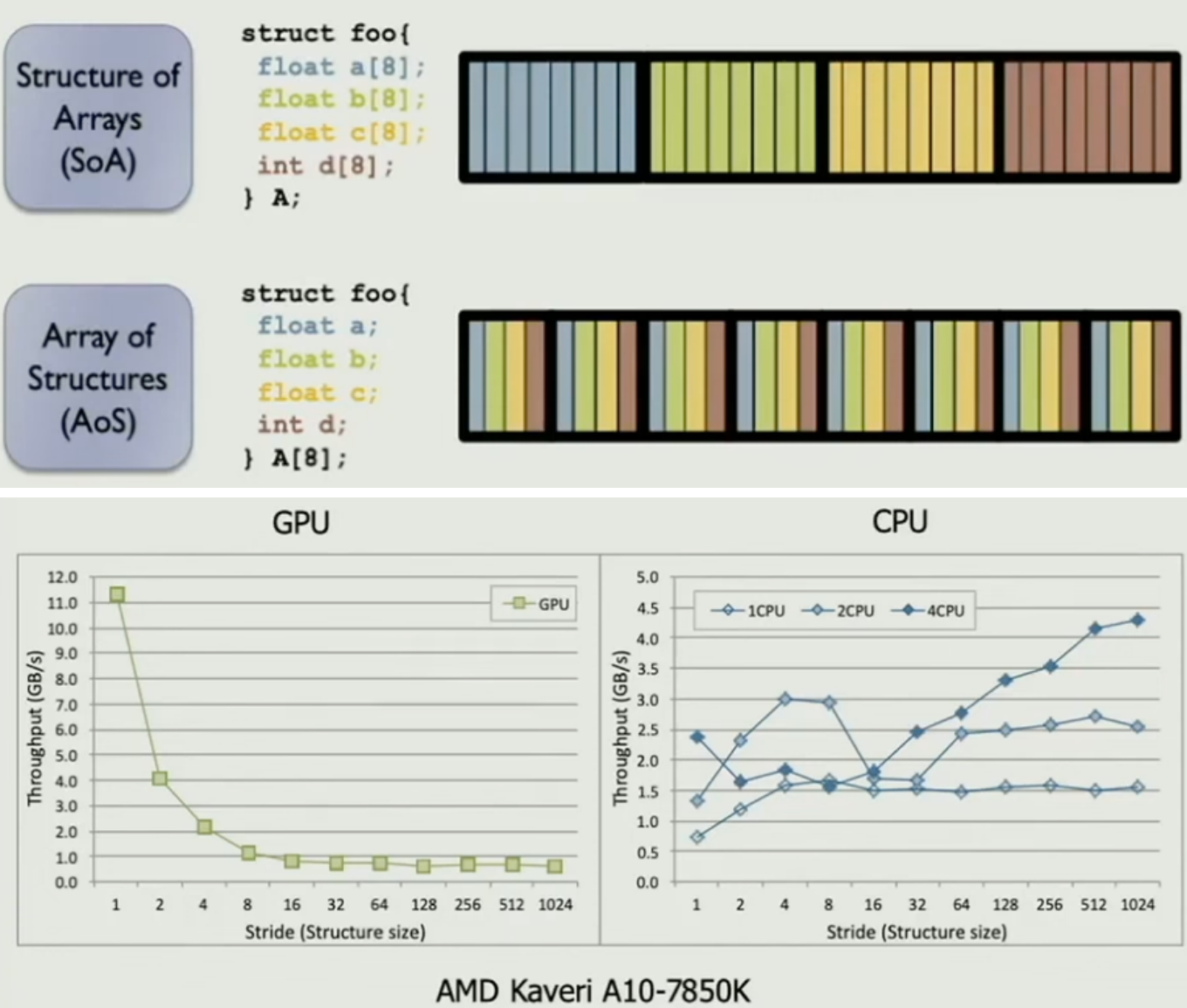


> T stands for thread
>
> Time period stands for Iteration

This causes 1 memory transaction to load the cache line.

AoS (Array of Structures) vs SoA (Structure of Arrays)

- **AoS (Array of Structures)**: Commonly preferred in CPU applications due to the locality of reference and data organization that fits well with the CPU cache line structure.
- **SoA (Structure of Arrays)**: More advantageous for GPUs as it aligns with the need for memory coalescing. By organizing data into SoA, each thread accesses a contiguous segment of memory, which is optimal for GPUs and helps in maximizing the memory bandwidth.



Memory access - Data reuse

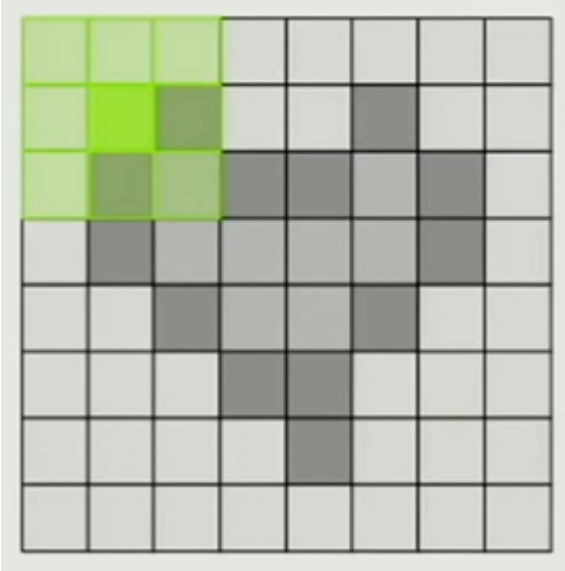
Particularly useful in applications like image processing, where the same data elements are accessed multiple times by different operations or iterations. Optimizing data reuse can significantly reduce the demand on memory bandwidth and improve overall performance.

- **Efficient memory use** involves accessing the same memory locations multiple times before discarding them. This is particularly efficient when neighboring threads access overlapping data segments, as in convolutional operations used in image processing and neural networks.

Example

Consider a typical filter operation in image processing, such as applying a Gaussian blur. Each pixel's new value is calculated using a combination of its own value and the values of its neighbors.

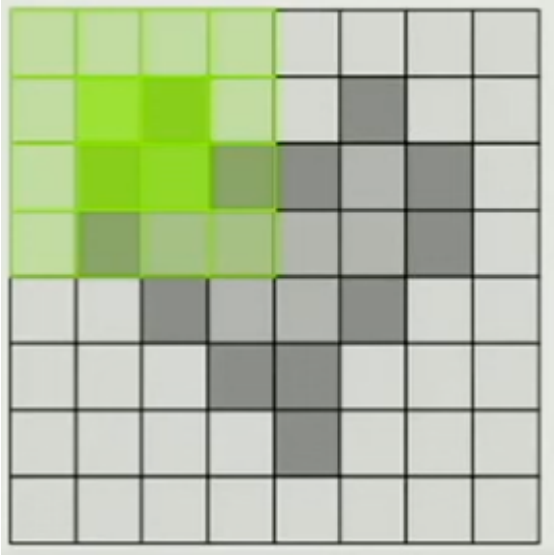
```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```



> Each thread computes a value based on a 3x3 region of the image. As the filter moves across the image, many of these regions will overlap, meaning subsequent operations will reuse some of the same data elements.
>
> What about the next 9 elements? We're going to access the same 6 elements again. This is data reuse. We do not want again to access the global memory to get the same data, if we can reuse it we can optimize the performance of the program. Moving the filter one pixel to the right in a row means six out of the nine pixels in the new filter window are the same as in the previous window.

Optimization techniques for data reuse

- **Shared Memory Utilization**: Temporarily store data that will be reused in the shared memory of the GPU, which is much faster to access than global memory. This reduces latency and bandwidth usage on repeated accesses.
- **Tiling** by breaking down the image or data array into smaller blocks or tiles that fit into shared memory. Process each tile independently while maximizing the reuse of data loaded into the shared memory.
- **Plan Access Patterns:** Organize thread access patterns so that threads access shared memory efficiently, avoiding bank conflicts and ensuring coalesced access to global memory when loading and storing data.



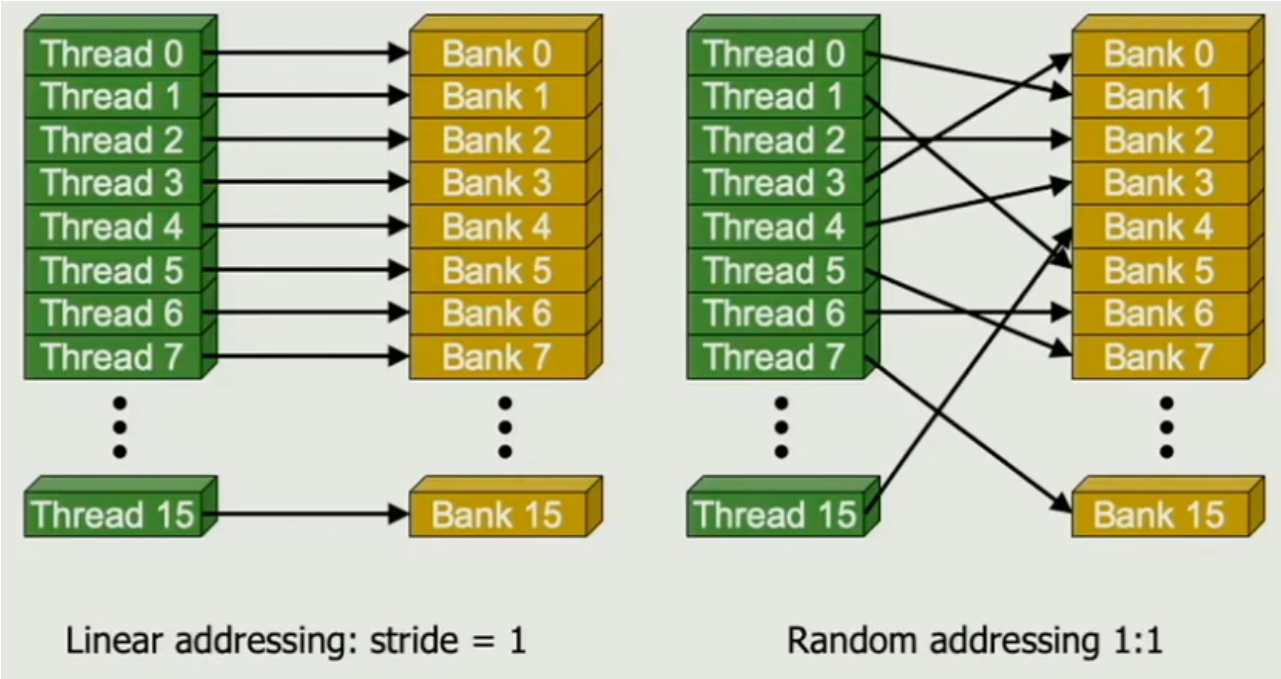
```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
// ..
// load tile into share memory
__syncthreads(); // make sure that all threads have finished loading the
data!
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2) + (j+l_col-1)];
    }
}
```

Shared memory

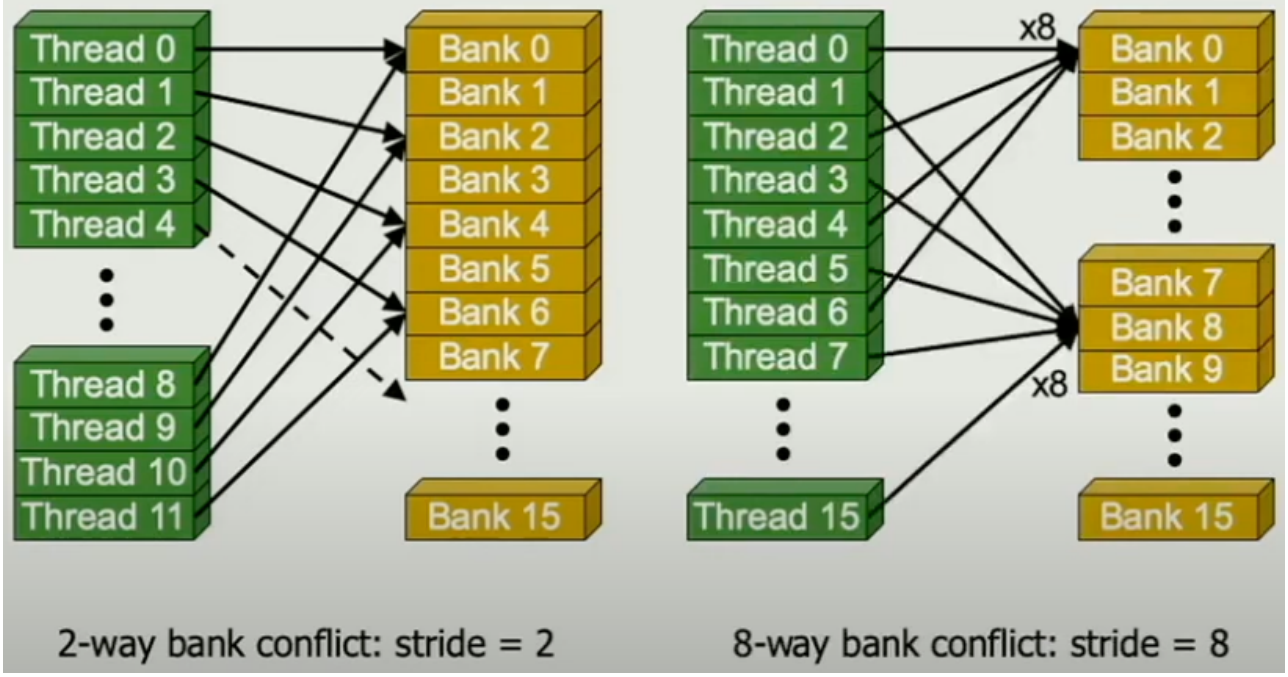
- Shared memory also had limits
- Is organized into multiple memory banks to allow concurrent access by multiple threads
- Typically, 32 banks in NVIDIA GPUs:
    - Successive 32-bit words are stored in successive banks
    - `Bank = Address % 32`
    - In an ideal scenario, each thread in a warp accesses a different bank, allowing all 32 threads to access memory concurrently without any delays.
- **Within-Warp Conflicts:** Bank conflicts occur when multiple threads in the same warp request data from the same memory bank simultaneously. This results in serialization of these memory accesses, degrading performance.
    - If each thread accesses a separate bank, there are no conflicts, and access is maximized.
    - If N threads access the same bank, they are serialized, effectively reducing the bandwidth by a factor of N.
- Bank conflicts are only possible within a warp

    - No bank conflicts between different warps, why? Because the access to the shared memory is scheduled at different times for different warps, whereas in the same warp, the access can be scheduled at the same time making the threads vulnerable to bank conflicts.

- Bank conflict free



- N-way bank conflicts



Reducing shared memory bank conflicts

- Bank conflicts are only possible within a warp
    - No bank conflicts between different warps. Different warps access shared memory at different scheduled times, thus not conflicting with each other.
- If strided accesses are needed, some optimization techniques can help

- **Padding**: Altering the data structure to add extra elements can prevent multiple threads from accessing the same bank. This technique is useful when data structures are statically sized and indexed.
- **Randomized mapping**: Changing the order of elements in shared memory or using a different addressing scheme can help distribute memory accesses more evenly across the banks.
- **Hash functions**: Applying a hash function to compute the address can randomize accesses across banks, reducing the likelihood of conflicts.

Drawbacks of shared memory

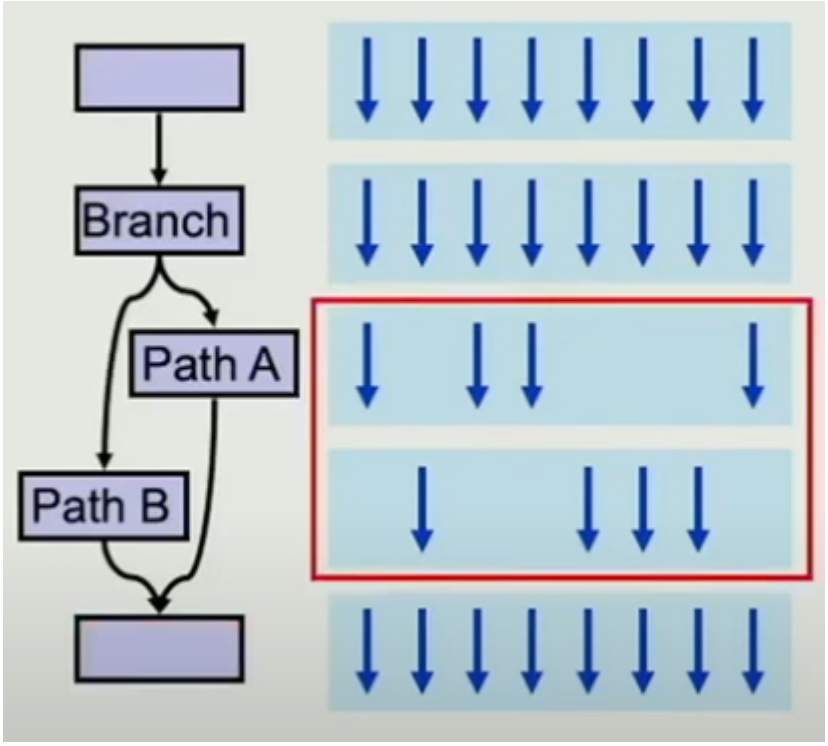
- Optimizing shared memory requires a deep understanding of memory access patterns within kernels. Programmers must anticipate how threads will access memory and adjust the data layout or indexing accordingly.
- Effective optimization often requires knowledge about the number of banks and the specifics of the GPU architecture, which might not always be readily available or could change with new GPU generations.

SIMD utilization

> Also seen here

Efficiency can be compromised by control flow divergence among threads within a warp.

- **Branch Divergence** occurs when threads within the same warp need to execute different execution paths based on their data. It leads to serial execution of divergent branches, reducing the effective utilization of GPU resources.



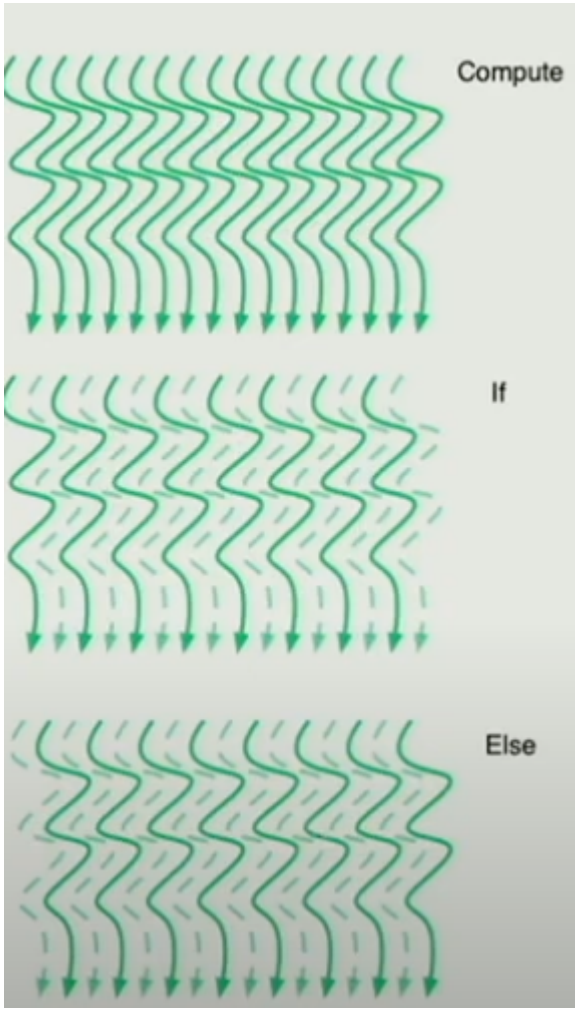
> In the image above causes 2 different execution paths, resulting in the twice the number of cycles to execute the code.

- **Hardware solutions** do exists, some advanced microarchitectures in CPUs mitigate branch divergence using techniques like branch prediction and speculative execution, GPUs typically do not implement these due to their design focused on massive parallelism and throughput.
- **Software Approaches** use software in a way that minimizes divergence is key. This involves understanding the common execution paths and structuring code to align with these paths as much as possible.

Example - Intra warp divergence

How threads within a warp can diverge based on simple conditions, leading to different functions being called based on thread indices

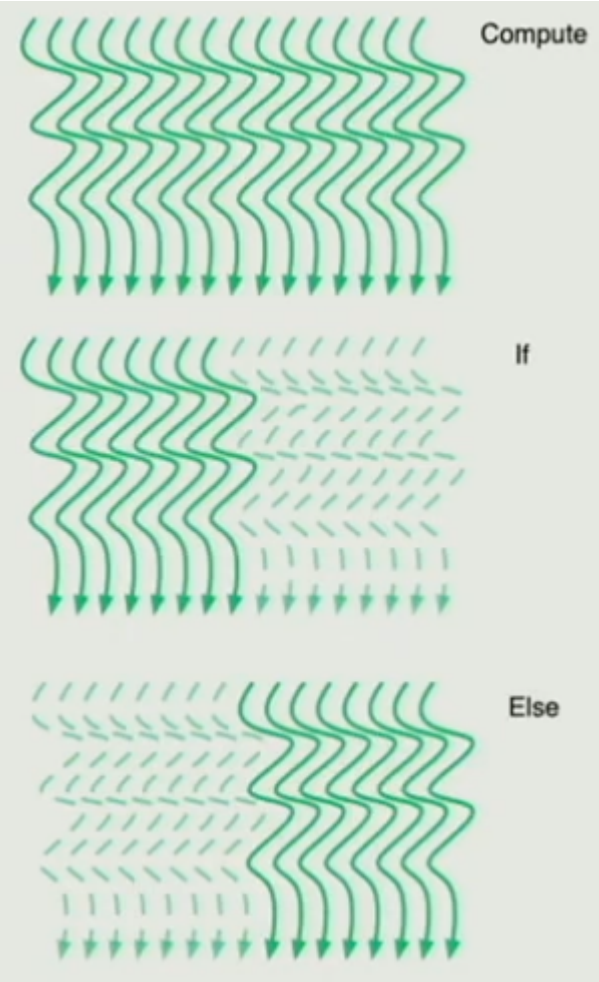
```
Compute(threadIdx.x);

if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
} else {
    Do_that((threadIdx.x % 32) * 2 + 1);
}
```



Example - Divergence-free execution

In this scenario, if the condition splits threads from different warps, each set of threads can execute concurrently without causing intra-warp divergence, thus maintaining full utilization of the SIMD unit.

```
Compute(threadIdx.x);

if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
} else {
    Do_that((threadIdx.x % 32) * 2 + 1);
}
```



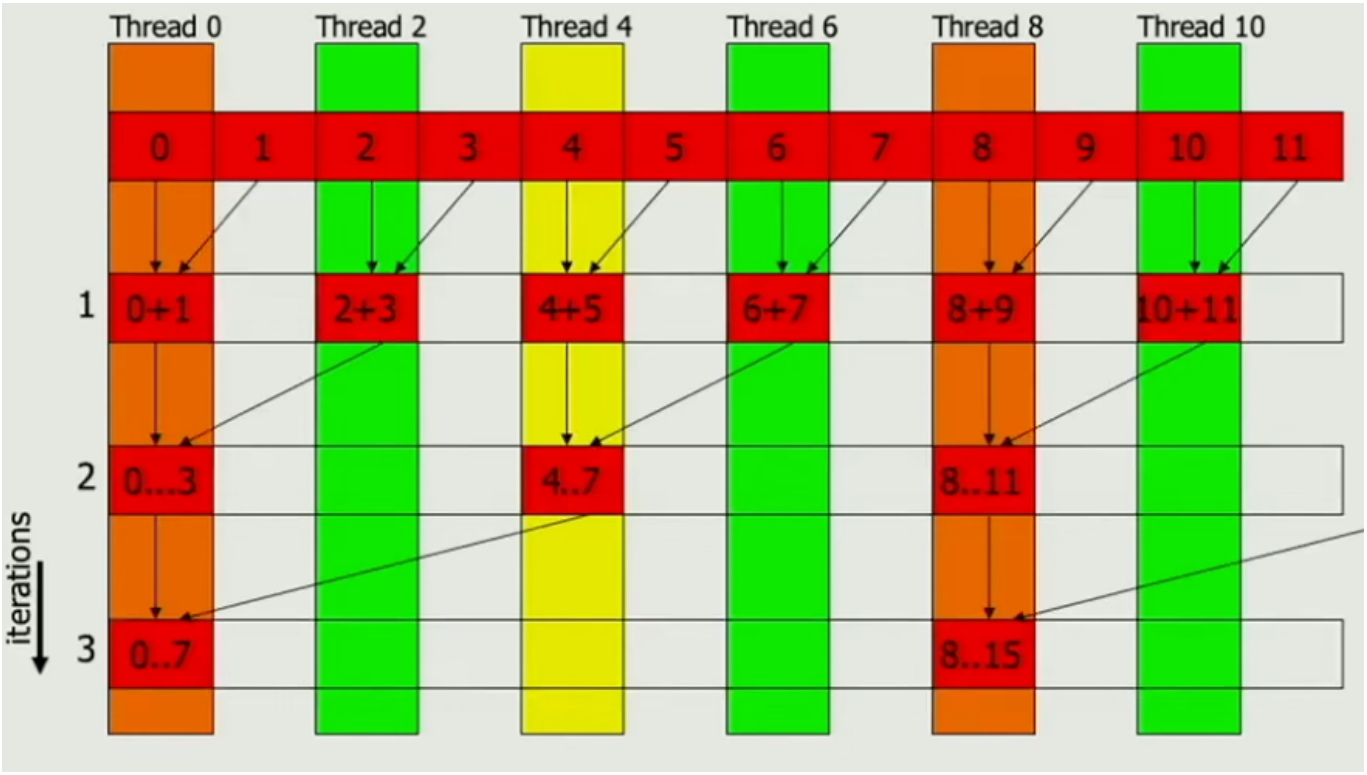
> If these conditional branches are handled by separate warps, there is no impact on the utilization within a single warp, ensuring efficient SIMD execution.

Vector reduction

Is a common parallel algorithmic pattern where all elements of an array are combined to produce a single value, such as the sum, minimum, or maximum. Efficiently implementing vector reduction on GPUs can significantly impact the performance of many applications.

Naive mapping

In this method, threads reduce pairs of elements in multiple steps, where each thread combines two elements and writes back the result. The number of active threads halves in each subsequent iteration, leading to poor SIMD utilization as fewer threads remain active.



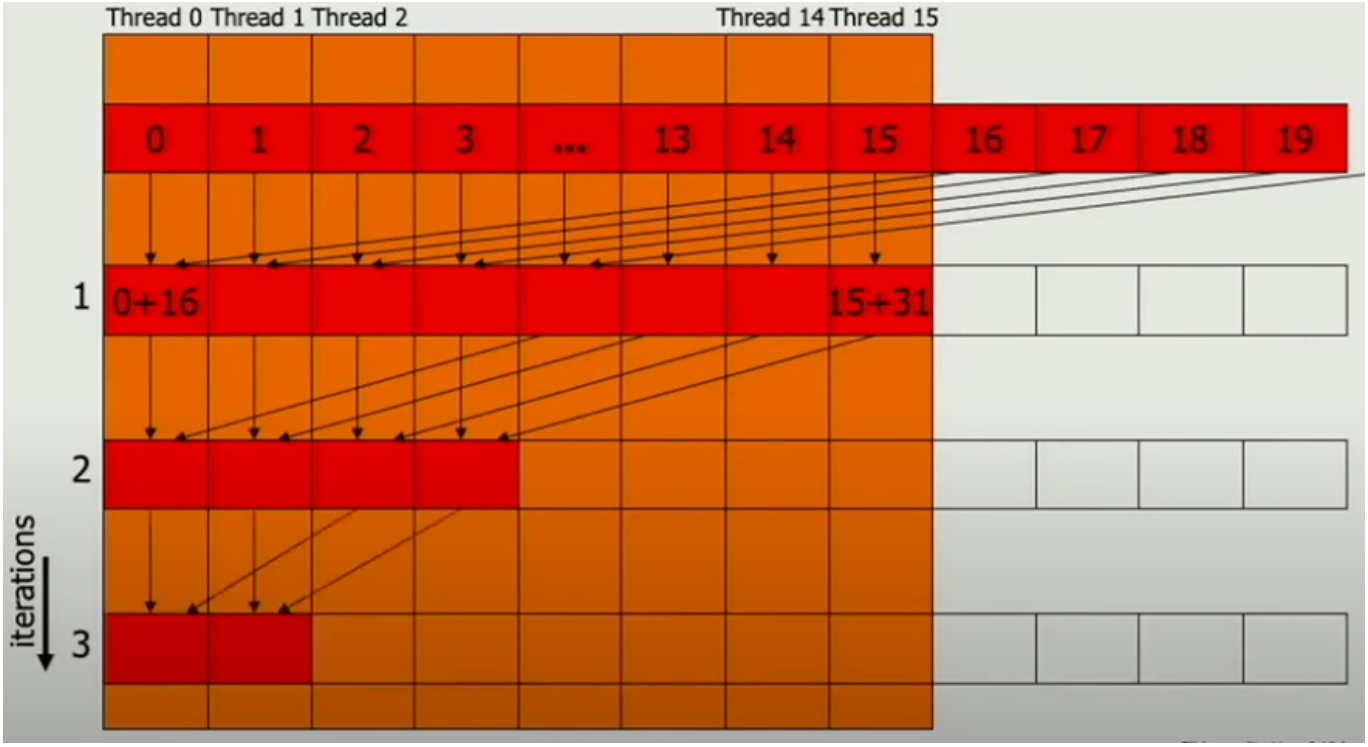
> This image illustrates how threads become increasingly idle in later iterations, as they have fewer elements to combine.

```
__shared float partialSum[]

unsigned int t = threadIdx.x;

for (unsigned int stride = 1; stride < blockDim.x; stride *= 2){
    __syncthreads();

    if (t % (2*stride) == 0){
        partialSum[t] += partialSum[t + stride];
    }
}
```

> This code demonstrates a typical reduction pattern where each thread progressively reduces elements spaced by increasing strides. However, this leads to increasing divergence and underutilization of the GPU's SIMD capabilities.

Divergence-free mapping

To improve SIMD utilization, it is preferable to ensure that all threads in a warp remain active without divergent execution paths. This approach modifies the loop to decrease the range of active threads more gradually and maintains more consistent thread activity.



> No 100% SIMD utilization, but better than the naive mapping.
>
> The code is a little bit more complex, but it's worth it.

```
__shared float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >> 1) {
    __syncthreads();

    if (t < stride) {
        partialSum[t] += partialSum[t + stride];
    }
}
```

> This code snippet uses a more sophisticated approach by halving the stride more smoothly, ensuring that the number of active threads decreases in a more controlled manner. It helps maintain a higher level of thread activity and reduces divergence.

Atomic operations

Atomic operations are crucial for ensuring correct computations when multiple threads need to update the same memory location concurrently. They are widely used in parallel algorithms for tasks such as counting, accumulating sums, or implementing mutexes.

- When multiple threads attempt to update the same memory location simultaneously, without atomic operations, data corruption or inconsistencies can occur.
- CUDA: `int atomicAdd(int*, int)`

- PTX: `atom.shared.add.u32 %r25, [%rd14], %r24`
    - PTX is an intermediate representation of the CUDA code that is generated by the compiler. Useful to ensure continuity of the code between generations of GPUs.
- SASS: Specific Assembly Code for the particular GPU architecture

```
; Tesla, Fermi, Kepler architectures
/*00a0*/ LDSLK P0, R9, [R8];
/*00a8*/ @P0 IADD R10, R9, R7;
/*00b0*/ @P0 STSCUL P1, [R8], R10;
/*00b8*/ @!P0 BRA 0Xa0;
```

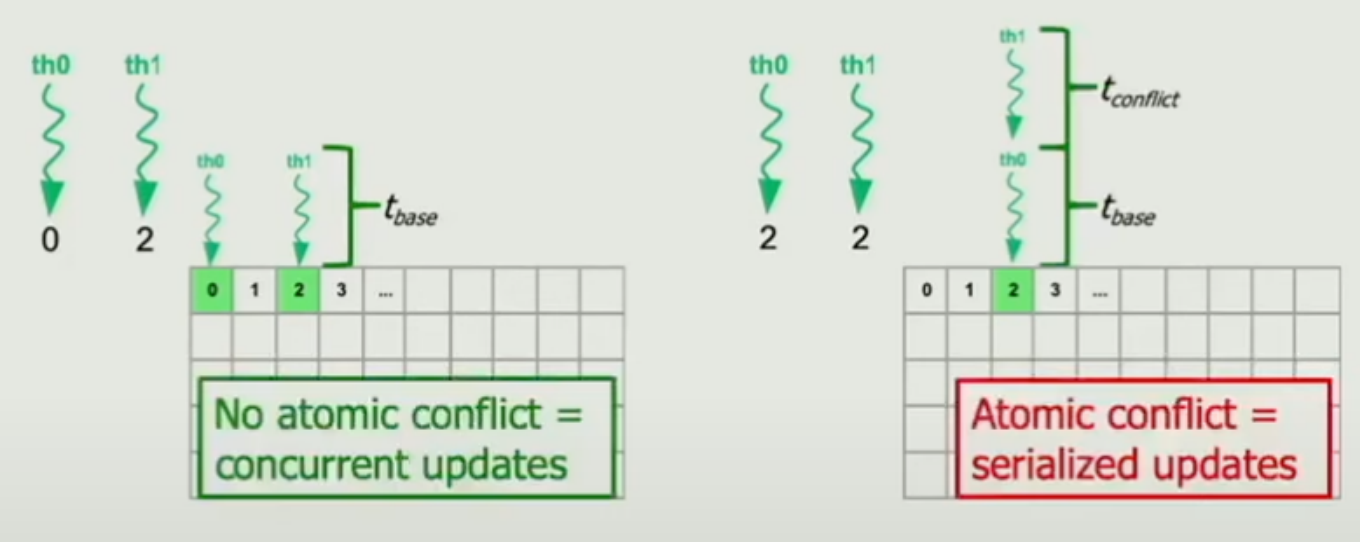
> The above was not very efficient. The atomicAdd where compiled to a load, add, store, and a branch instruction. P0 & P1 (predicate registers) are 1 **bit per thread**, 0 or 1 depending on whether the thread managed to acquire the lock or not. So if 2 threads belonging to the same warp wanted to write to the same memory location, they have to contend for this lock, and only one of them will be able to write to the memory location. The thread that did not manage to acquire the lock would have retried the operation. -> **VERY INEFFICIENT** when multiple threads tried to write to the same memory location.

> Newer GPU generations introduced more efficient atomic operations that reduce the need for locks and retries, thanks to hardware-level support for atomicity.

```
; Maxwell, Pascal, Volta
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

> Native atomic operations for 32-bit integer and 32-bit and 64-bit atomicCAS (compare and swap) operations.

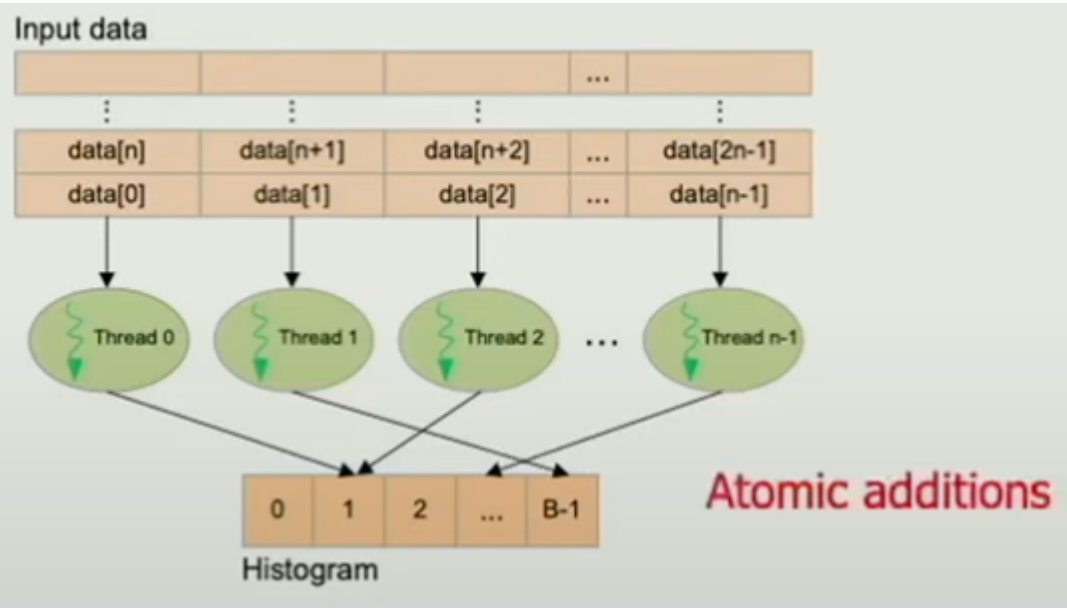
- **Intra-Warp Conflict Degree** is the number of threads within a warp that try to update the same memory position can significantly impact performance. The conflict degree can range from 1 (no conflict) to 32 (all threads in a warp update the same location).
- **Serialization and Performance:** high conflict degrees lead to serialization of updates, which can severely degrade performance.



Histogram calculation

Histograms count the number of data instances in disjoint categories (bins)

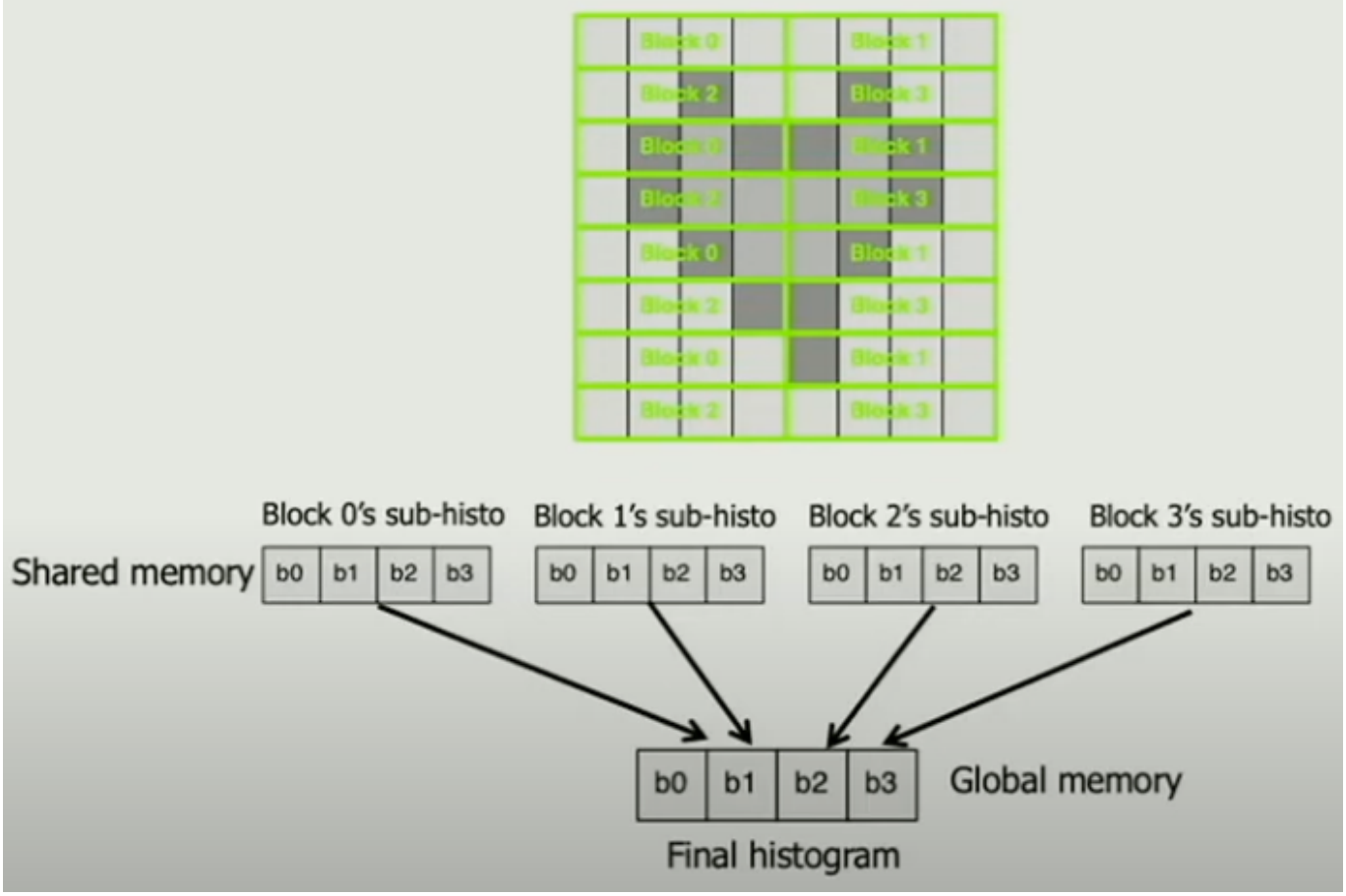
```
for (each pixel i in image I) {
    Pixel = I[i]                // Read pixel
    Pixel`= Computation(Pixel) // Optional computation
    Histogram[Pixel`]++         // Vote in histogram bin
}
```



- Frequent conflicts in natural images because of the high number of pixels with the same value

Optimizing the histogram calculation

- **Privatization**: Per-block sub-histograms in shared memory, basically, instead of using global memory, we use shared memory to store the histogram. Each block has its own histogram in shared memory, and at the end, we combine all the histograms from all the blocks.



Data transfers between CPU and GPU

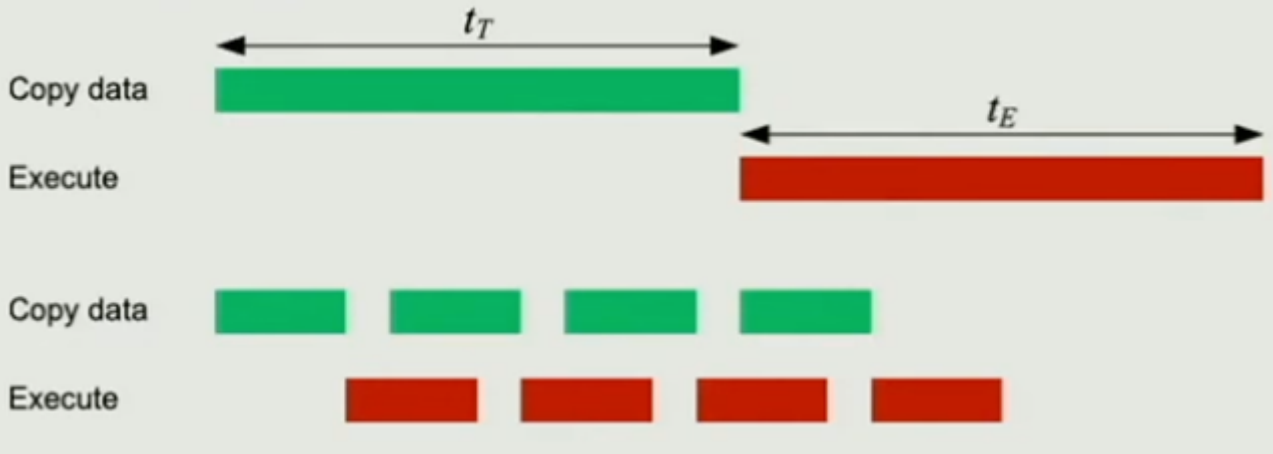
- One of the main bottlenecks in GPU computing
- Glued together by the **PCIe bus** or **NVLink**
    - These are the primary channels for data transfer between the CPU and GPU. While NVLink offers significantly higher bandwidth than traditional PCIe connections, the bandwidth (ranging from 300 to 900 GB/s) can still be a limiting factor in overall application performance.

Types of Data Transfers

- **Synchronous Transfers**: Operations like `cudaMemcpy()` block the CPU until the GPU finishes the data transfer. This can lead to inefficiencies if the CPU or GPU has to wait for the other to complete its tasks.
- **Asynchronous Transfers**: `cudaMemcpyAsync()` allows the CPU to continue processing other tasks while the GPU handles the data transfer, potentially overlapping with other GPU operations like kernel execution.
- **Streams:** These are sequences of operations that the GPU performs in order. Using multiple streams can greatly enhance the efficiency of data transfers and kernel executions by organizing them into independent sequences that can be processed concurrently.

Asynchronous data transfers

- Computation divided into nStreams
  - D input data instances, B blocks
  - nStreams
    - D/nStreams data instances
    - B/nStreams blocks



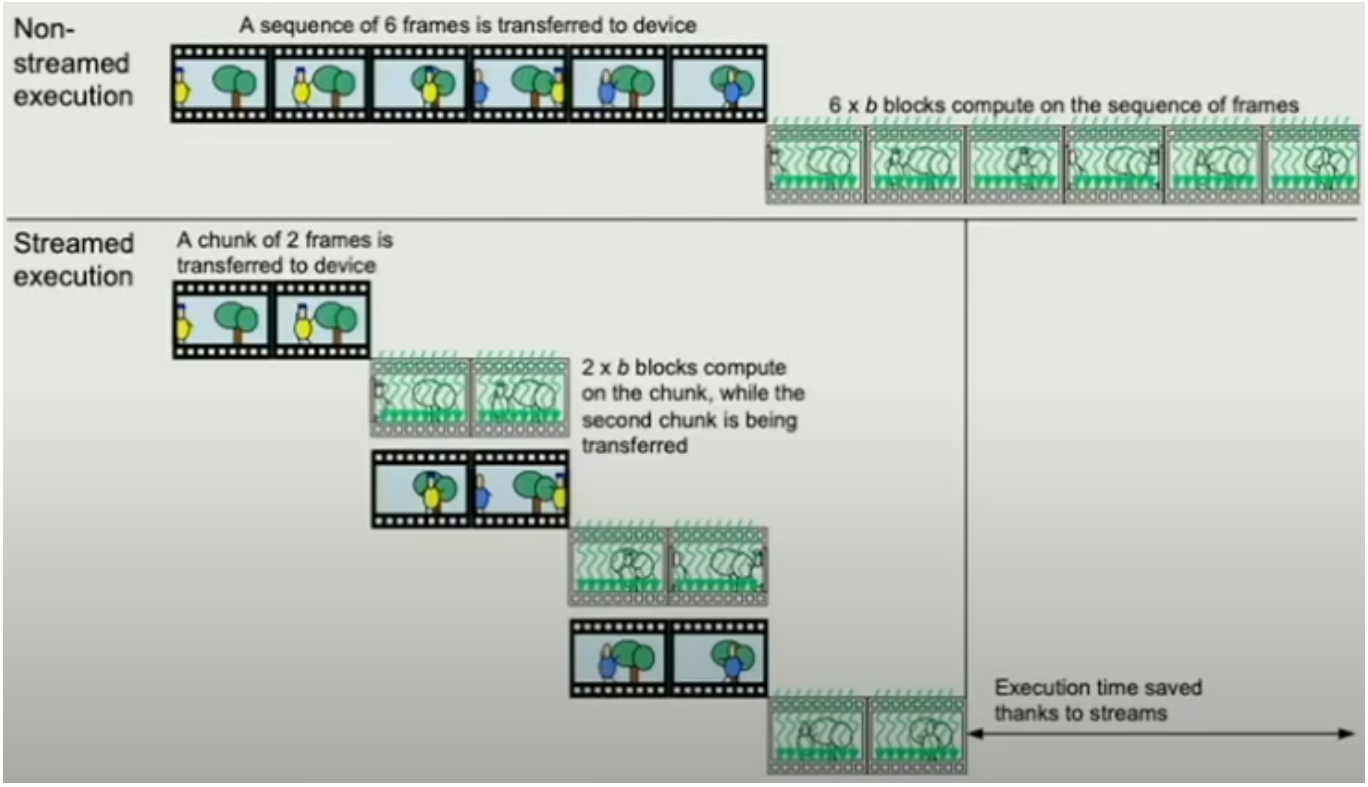
> Not always possible

- Estimates
  - `t_e + t_T / nStreams`, where `t_e >= t_T (dominant kernel)`
  - `t_T+ t_e / nStreams`, where `t_T >= t_e (dominant transfers)`

> Little overhead added in managing the streams, but worth it.

Example - Video processing

- Applications with independent computation on different data instances can benefit from asynchronous transfers



Unified Memory

- simplifies memory management by allowing the CPU and GPU to share the same memory space, eliminating the need for explicit data transfers.
- Particularly beneficial in applications where frequent data exchange between the CPU and GPU is necessary. It reduces the overhead of managing separate memory spaces and can lead to simpler and more efficient code.

Summary

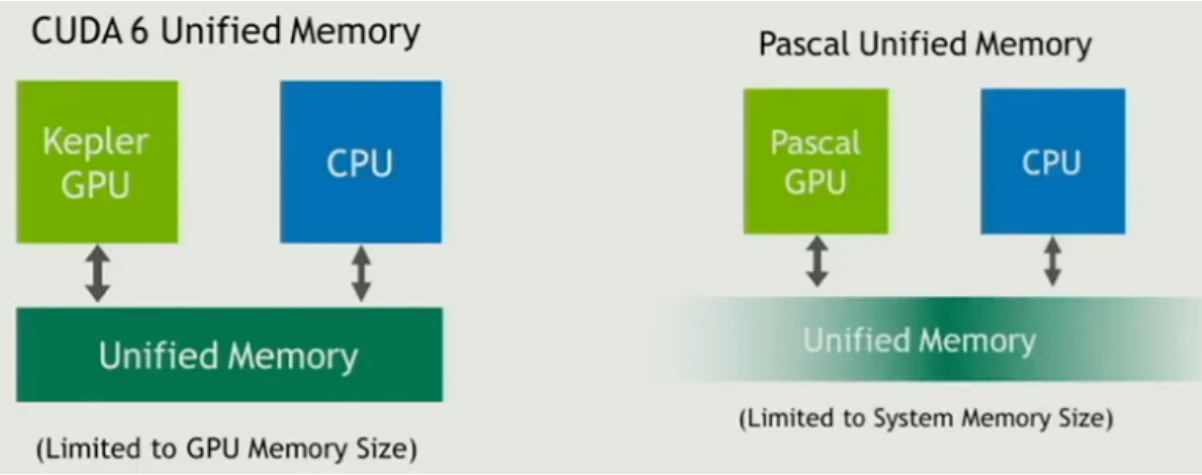
- GPU as an accelerator
    - Program structure
        - Bulk synchronous programming model
    - Memory hierarchy and memory management
    - Performance considerations
        - Memory access
            - Latency hiding: occupancy (TLP)
            - Memory coalescing
            - Data reuse: shared memory
        - SIMD utilization
        - Atomic operations
        - Data transfers

Collaborative computing

Collaborative computing between CPUs and GPUs can be complex due to the fundamentally different architectures and processing capabilities of each. However, advancements like Unified Memory and asynchronous kernel launches have significantly improved the synergy between these two components.

Unified Memory

- The CPU and GPU share the same memory space. This eliminates the need for manual data transfers and helps manage data consistency automatically.
    - Since CUDA 6.0: unified memory
    - Since CUDA 8.0 + Pascal architecture: GPU page faults



- Easier programming
    - Developers no longer need to explicitly copy data between the CPU and GPU. Memory allocation with `cudaMallocManaged()` automatically ensures that the data is accessible from both the CPU and GPU.
    - No need to manage data consistency which reduces the risk of errors and simplifies the code.

Asynchronous kernel launches

- **Non-Blocking Computations**: Kernel launches are asynchronous, meaning the CPU can perform other tasks while the GPU is processing. This allows for more efficient use of system resources.

```
// allocate input
malloc(input, ...);
cudaMallocManaged(&d_input, ...);
memcpy(d_input, input, ...);

// allocate output
malloc(output, ...);
cudaMallocManaged(&d_output, ...);

// launch kernel
gpu_kernel<<<...>>>(d_input, d_output);

// CPU can do things here!

// sync
cudaDeviceSynchronize();

// copy output to host memory
cudaMemcpy(output, d_output, ...);
```

Fine-grained heterogeneity

With architectures like Pascal and Volta, more sophisticated features such as

- **CPU-GPU memory coherence** which means that if the GPU updates a memory location, the change is immediately visible to the CPU and vice versa, without the need for explicit synchronization commands.
- **System-wide atomics** are supported which extend the concept of atomic operations to work across the entire system, meaning that these operations can be performed with consistency and atomicity no matter whether they are initiated by the CPU or the GPU.

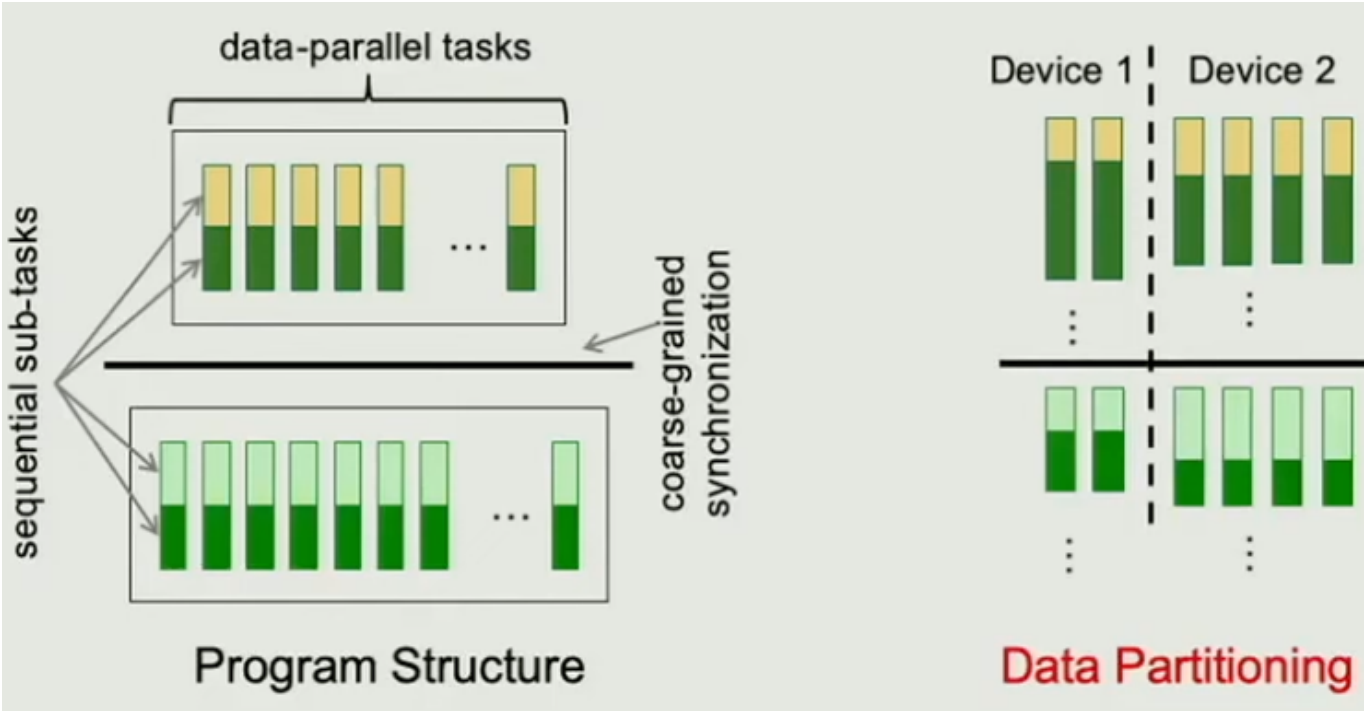
Allows for fine-grained control over how data is accessed and modified, enhancing the capabilities for collaborative computing.

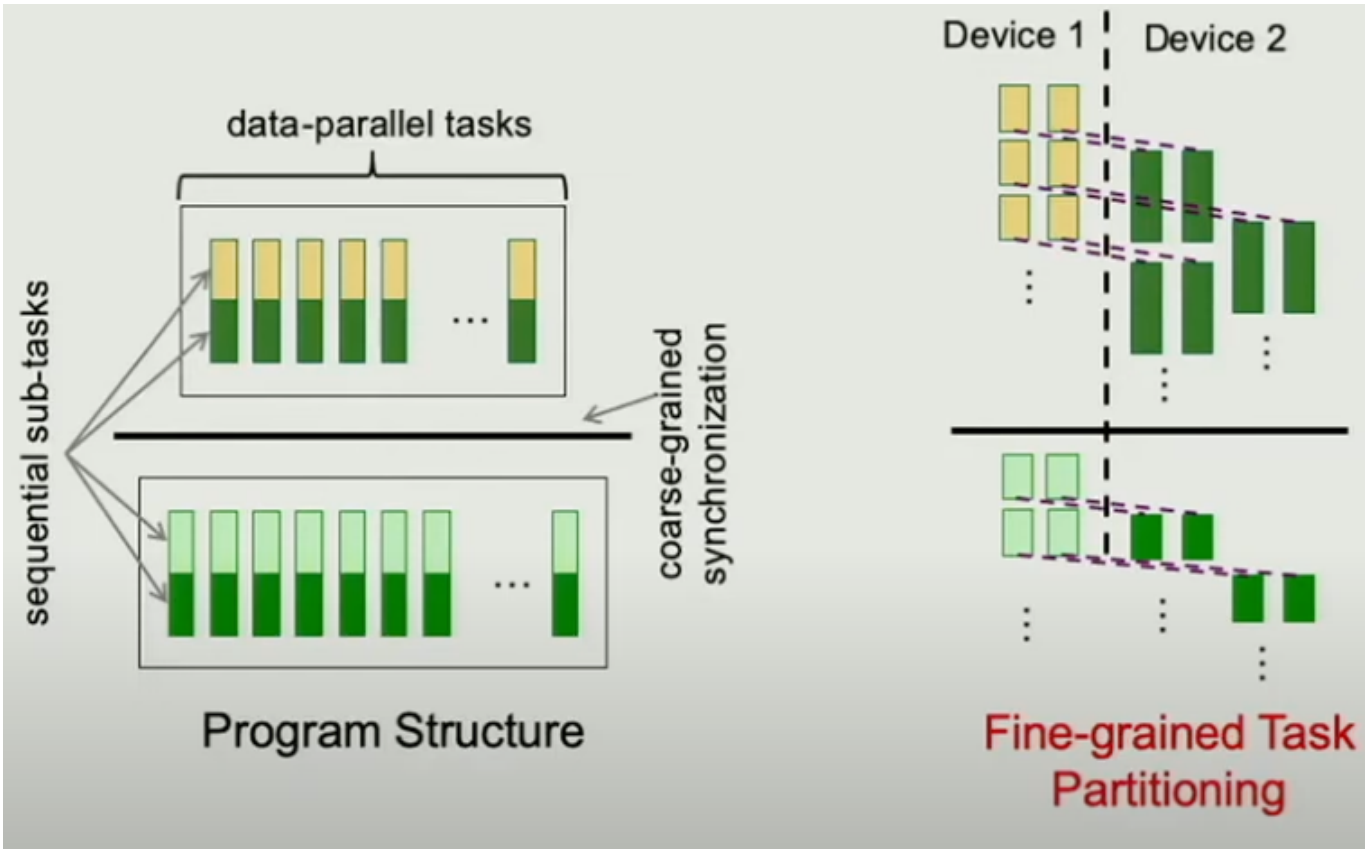
- **Reduced Complexity** so that developers no longer need to implement complex data transfer and synchronization mechanisms in their applications, simplifying code and reducing potential bugs.
- **Performance Enhancements** by reducing the need for data transfers and manual synchronization can significantly speed up applications, particularly those involving frequent interactions between the CPU and GPU.
- **Enhanced Flexibility** so that applications can more dynamically distribute tasks between the CPU and GPU based on their respective strengths and current load.

```
// allocate input
cudaMallocManaged(&d_input, ...);

// allocate output
cudaMallocManaged(&d_output, ...);

// launch kernel
gpu_kernel<<<...>>>(d_input, d_output);

// CPU can do things here!
output[x] = input[y];

output[x+1].fetch_add(1);
```

> Since CUDA 8.0
>
> - Unified memory `cudaMallocManaged(&h_in, in_size)`
> - System-wide atomics `old = atomicAdd_system(&h_out[x], inc)`

Collaborative patterns





22_CUDA

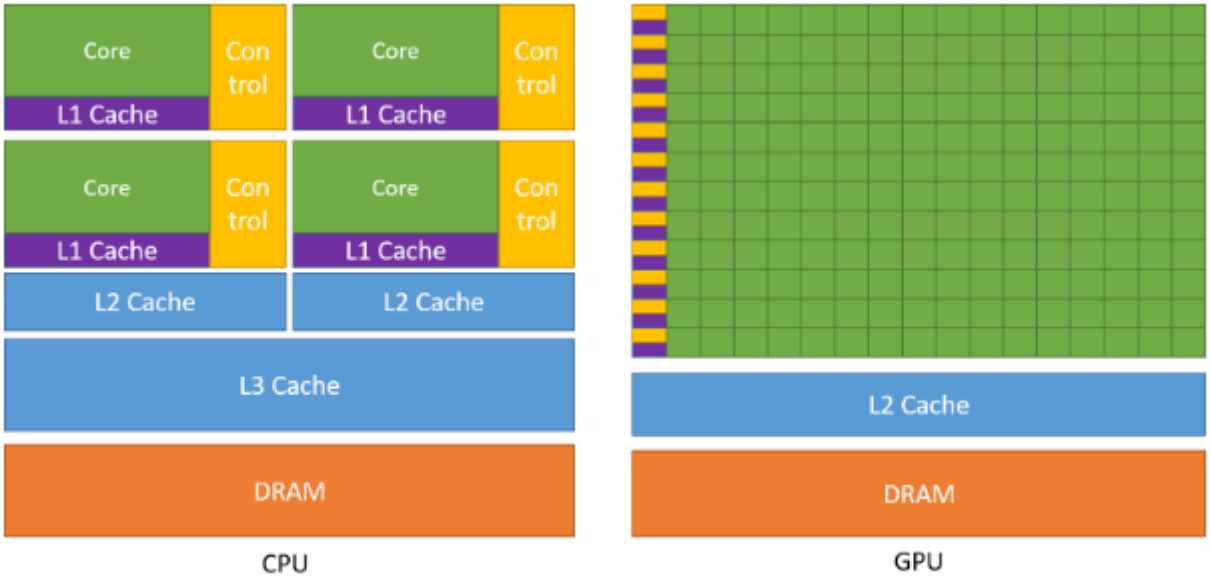
> https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Why GPUs?

- GPUs are designed to have a high throughput for parallel workloads.
- While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at

executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

- The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control



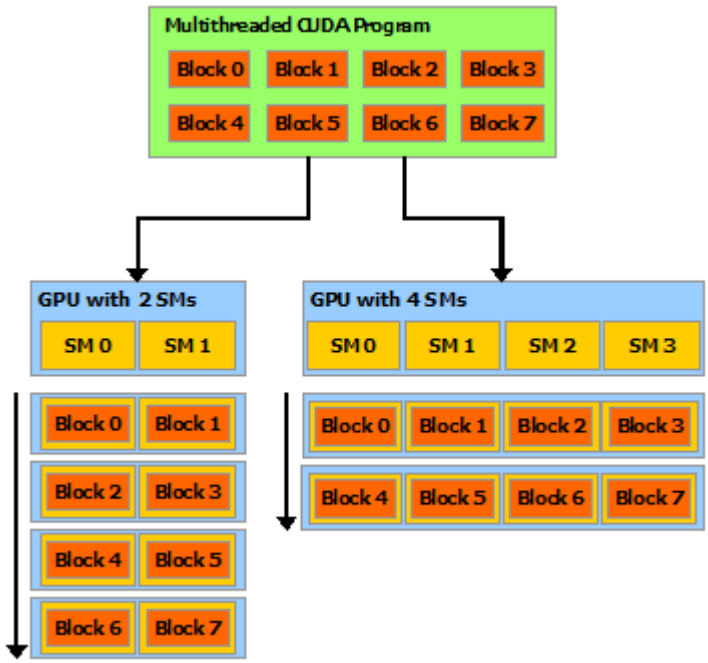
- More FP computations -> higher throughput for parallel workloads
- GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.
- The GPU and CPU are designed to work together to provide the best performance for a wide range of applications.
- Applications are a mix of serial and parallel workloads

CUDA (Compute Unified Device Architecture)

- General purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve complex computational problems.
- C/C++ extension

CUDA Programming Model

- Since is an extension of C/C++, it is easy to learn
- At its core it exposes the following to programmers:
  - Hierarchical thread hierarchy
  - Shared memory
  - Barrier synchronization
- The abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism.
- The programming model guides the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can solved cooperatively by the threads in the block.



Kernels

- Kernels are simply functions that are executed on the GPU.
- When called are executed N times in parallel by N different CUDA threads, as opposed to only once like a regular C/C++ function.
- Each thread that exectues the kernel is given a unique threadID that is accessible within the kernel code.

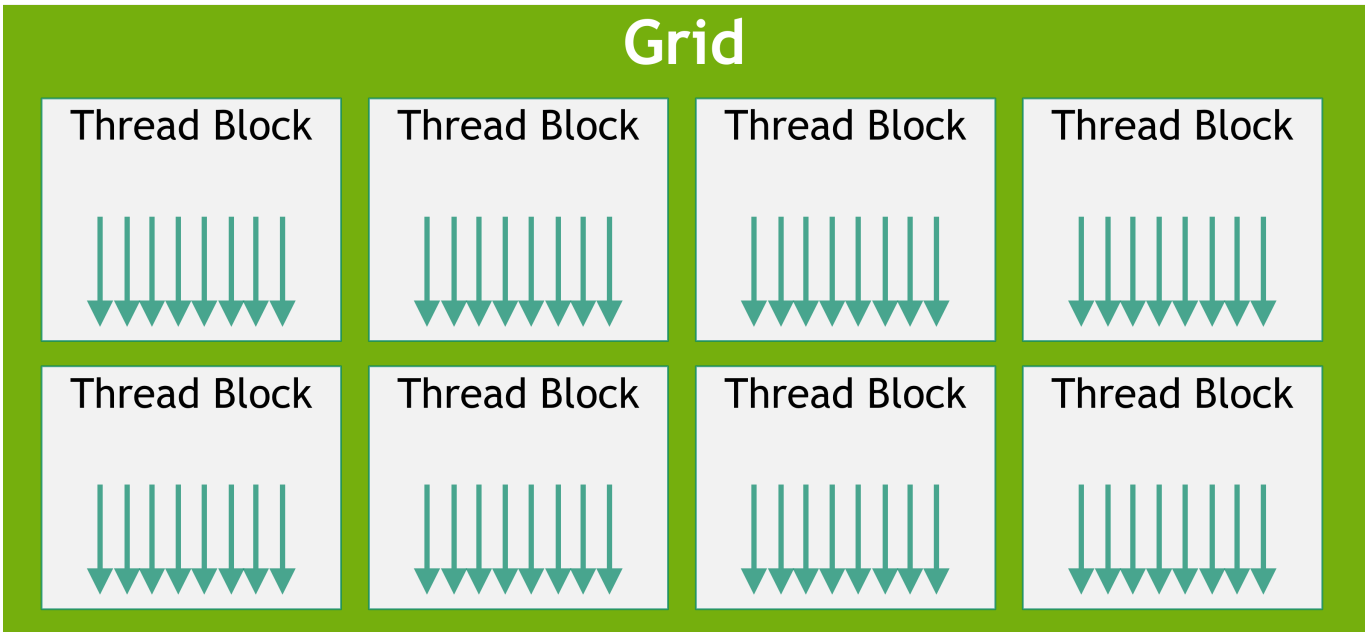
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

> As an illustration, the following sample code, using the built-in variable threadIdx, adds two vectors A and B of size N and stores the result into vector C:
>
> A kernel is defined using the **global** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax

Thread Hierarchy

```
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- The index of a thread and its thread ID relate to each other in a straightforward way:
  - one-dimensional blocks are laid out in a single line
  - two-dimensional blocks are laid out in a grid (Dx, Dy), where the thread ID of a thread of index `(x, y)` is `x + y * Dx`
  - three-dimensional blocks are laid out in a 3D grid (Dx, Dy, Dz), where the thread ID of a thread of index `(x, y, z)` is `x + y * Dx + z * Dx * Dy`
- There's a limit to the number of threads per block, all threads of a block are executed on the same SM, and all threads of a block can communicate with each other through shared memory.
- A thread block may contain up to 1024 threads
- A kernel can be executed by multiple equally-sized thread blocks

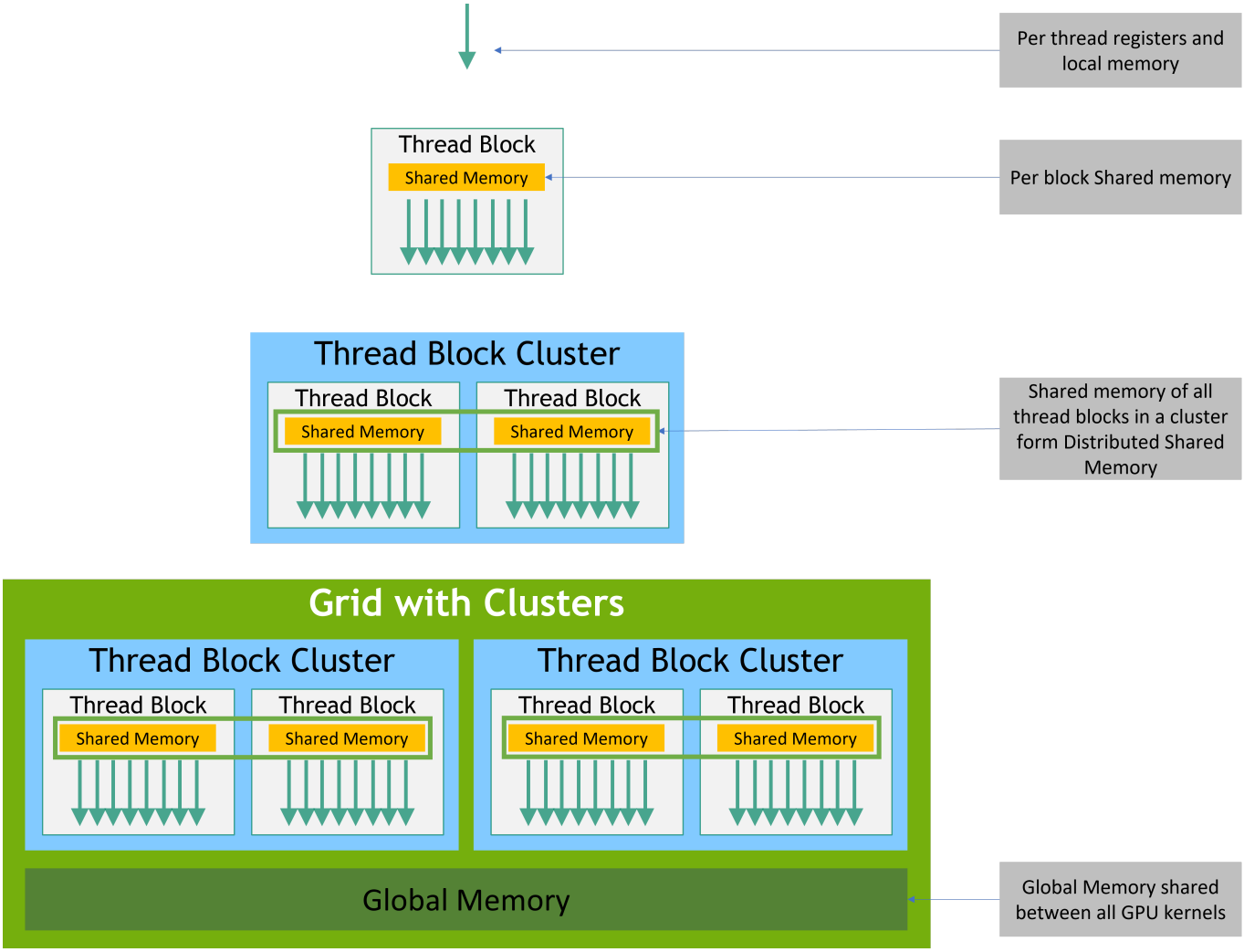


- Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series.

- Threads within a block can cooperate through shared memory, and threads within a grid can cooperate through global memory.
- The synchronization barrier `__syncthreads()` can be used to synchronize threads within the same block.
- When using the function, all threads within the block must reach the same point in the code before any is allowed to proceed.

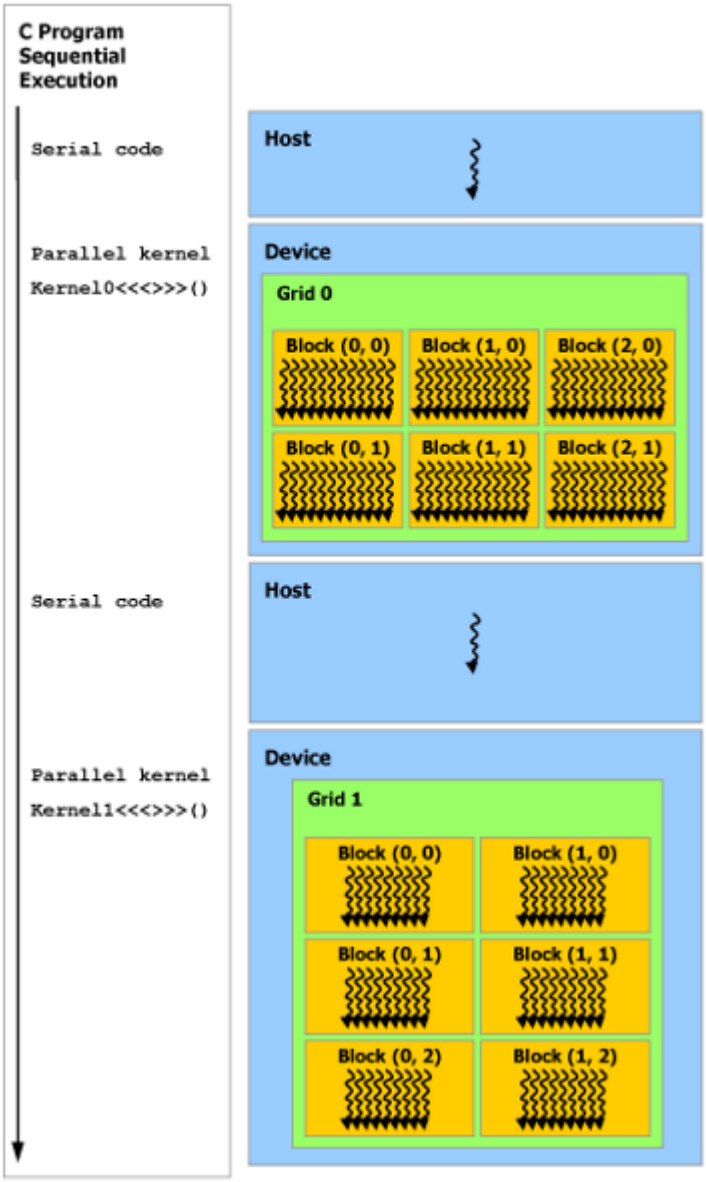
Memory Hierarchy

- Each thread has a private local memory
- Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block
- All threads have access to the same global memory



Heterogeneous Programming

- Modern applications are a mix of serial and parallel workloads
- The CPU is good at serial workloads
- The GPU is good at parallel workloads
- Each retain their own separate memory spaces in DRAM, referred to as the host memory and device memory, respectively



> Serial code executes on the host (CPU) and parallel code executes on the device (GPU)

Unified memory

- Introduced in CUDA 6
- Allows the CPU and GPU to access the same virtual memory space
- Simplifies memory management and reduces the need to explicitly copy data between the CPU and GPU
- Basically a single pointer value enables all processors in the system (all CPUs, all GPUs) to access this memory with all of their native memory access instructions
- Pros:
    - Simplifies memory management
    - **Productivity**: Easier to write correct code. GPU programs may access Unified Memory from GPU and CPU threads concurrently without needing to create separate allocations (`cudaMalloc()`) and copy memory manually back and forth (`cudaMemcpy*()`).

    - **Performance**:
        - Data access speed may be maximized by migrating data towards processors that access it most frequently
        - Total system memory may be reduced by avoiding duplicating memory on both CPUs and GPUs
    - **Functionality**: enables GPU programs to work on data that exceeds the GPU memory capacity

- Data movement still takes place

Asynchronous SIMT Programming Model

- The SIMT (Single Instruction, Multiple Thread) programming model is a key feature of CUDA.
- A thread is the lowest level of abstraction for ding a computation or a memory operation.
- The asynchronous programming model defines the behavior of asynchronous operations in CUDA threads

Asynchronous Operations

- An asynchronous operation is defined as an operation that is initiated by a CUDA thread and is executed asynchronously as-if by another thread.
- The CUDA thread that initiated the operation may continue to execute other operations while the asynchronous operation is being executed.
- An asynchronous operation uses a synchronization object to synchronize the completion of the operation

CUDA Programming Interface

- The CUDA programming interface consists of a set of host functions and device functions.
- The compilation is do with the `nvcc` compiler driver
- The CUDA runtime API provides a simple interface for C and C++ functions that can be called from the host that manage the allocation, deallocation, and transfer of data between the host and the device.
- The runtime is built on top of a lower-level C API

Compilation with NVCC

- Kernels can be directly written in PTX (Parallel Thread Execution) assembly language, as a programmer would write directly assembly code for a CPU
- It is instead more common to write kernels in C/C++ and let the compiler generate the PTX code
- After the PTX code is generated, it is compiled to machine code by the PTX compiler named SASS which is architecture-specific

Workflow

- Source files compiled with nvcc can include both host and device code
- nvcc separates the host code from the device code and compiles each separately
    - Host code is compiled to object code by the host compiler
    - Device code is compiled to PTX code by the device compiler

CUDA runtime

- The runtime is implemented in a shared library that is linked with the application at runtime
- All entry points are prefixed with `cuda`