GPUs and Heterogeneous Systems

github.com/martinopiaggi/polimi-notes

2023-2024

Contents

1 Introduction to heterogeneous systems

An **heterogeneous System** comprises multiple computing units, each with distinct characteristics. Heterogeneous systems are necessary since the need to run applications in various fields with differing performance requirements (Email, browser, videogames , cad ... etc).

Main motivations are:

1. The need for high performance and energy efficiency in modern devices, which are **power-constrained**.
2. Many compute-intensive applications can be partitioned into parts with different characteristics, each of which can be efficiently accelerated by a different type of computing unit.

Challenges:

1. **Integration** of many different units and **communication** between them.
2. **Programmability**, as each type of unit has different programming languages and paradigms.
3. **Resource management**, including distributing a multi-programmed dynamic workload, configuring software/hardware knobs, and achieving required performance at a power-efficient level.

Real-world examples are infinite: mobile phones, laptops, desktop computers, embedded and edge devices, and supercomputers.

1.1 Evolution of Computing System Architectures

The evolution of computing system architectures can be summarized as:

- **Single-Core Era**: This era was characterized by increasing voltage/frequency scaling until the power wall was reached in 2004.
- **Multi-Core Era**: This era saw the integration of multiple cores in the same chip. However, performance limits were encountered due to power consumption and scalability issues.
- **Heterogeneous Systems Era**: marked by the integration of heterogeneous units in the same chip, allowing for parallelization of various applications. Different parts of the applications may benefit from specialized computing units.

A useful classification to remember:

Task Parallelism: This involves the execution of many independent tasks/functions in parallel. **Data Parallelism**: This involves operations on data composed of many items that can be processed simultaneously.

Flynn's taxonomy, introduced by Michael J. Flynn in 1966, and have been instrumental in understanding and developing parallel processing architectures:

- **SISD (Single Instruction, Single Data)**: Represents the classical von Neumann architecture where a single processing unit executes a single instruction stream on a single data stream. This is typical of traditional uniprocessor machines.
- **SIMD (Single Instruction, Multiple Data)**: Multiple processing units perform the same operation simultaneously on different data items. This architecture is often used in vector machines, where each processing unit executes the same instruction but on different pieces of data. Example: vector processors.
- **MISD (Multiple Instruction, Single Data)**: An uncommon architecture where multiple instructions operate on a single data stream. This type of architecture is rarely used in practice.
- **MIMD (Multiple Instruction, Multiple Data)**: Multiple autonomous processors simultaneously execute different instructions on different data items. This is common in multi-core machines where each core can run different instructions independently. Example: multi-core processors.
- **SPMD (Single Program, Multiple Data)**: Multiple autonomous processing units simultaneously execute a single program on different data items. While the program is the same across all units, the execution path can vary for each data point. This is the approach of CUDA:
  - The kernel function includes the code executed by each thread

  - For example a `for` loop is replaced by a grid of threads, each one working on a single data element
- **MPMD (Multiple Program, Multiple Data)**: Multiple autonomous processing units simultaneously execute at least two independent programs on different data items. This architecture allows for more complex and varied computation as different units can run entirely different programs.

1.1.1 Refresh of architectural solutions for parallel computing

In a computer architecture parallelism can be extracted at different levels:

- Instruction-Level Parallelism (ILP)
  - The architecture is organized in different stages
  - Each instruction "occupies" a single stage to permit overlapping between others
  - In a superscalar architecture (composed by several ALUs) instructions are scheduled out of order.
  - In a Very Long Instruction Word (VLIW) architecture Dependency analysis is performed by the compiler to schedule long instructions
- Data-Level Parallelism (DLP) (SIMD)
  - The architecture contains groups of several ALUs of the same type
  - Vectorized instruction are used to make the same operation executed on different data
- Thread-Level Parallelism (TLP):
  - Independent threads (no dependencies between each other) executes all together multiple instruction streams
  - The architecture stores the execution data (called context) of all the threads
  - The architecture contains several cores (multicore architecture) where each of them executes at least one thread.
  - A need for a cache hierarchy to avoid memory accesses to be the bottleneck.

Actually all these techniques are used in the GPUs but the most important is Data-Level Parallelism (DLP): where groups of ALUs execute the same operation on different data items simultaneously.

2 GPU architecture

The focus will be on NVIDIA architectures, but other vendors followed a similar path.

2.1 GPU evolution

The main steps in GPU Evolution can be resumed in:

1. GPU as a hardware accelerator for graphics pipeline
2. GPU for general-purpose computing
3. GPU architecture evolution to the present day

2.1.1 GPU as a hardware accelerator for graphics pipeline

The GPU is actual a general pathos accelerator. If you care about computer graphics, there is another course that is on computer graphics . The starting point here is to understand how the graphics pipeline works just because the has been originally defined based on this idea:

- **3D Rendering:** The process of converting a 3D model into a 2D image, performed in real-time or offline depending on the application
- **Graphics Pipeline for 3D Rendering:** A conceptual model describing the necessary steps for real-time 3D rendering, standardized around OpenGL/Direct3D APIs and possibly accelerated in hardware.

The real-time graphics pipeline implemented by OpenGL before 2007, consisting of a sequence of operations on several types of objects:

```
graph TD
    A[Vertex Generation] --> B[Vertex Processing]
    B --> C[Primitive Generation]
    C --> D[Primitive Processing]
    D --> E[Fragment Generation]
    E --> F[Fragment Processing]
    F --> G[Pixel Operations]

    subgraph "Real-time Graphics Pipeline"
        A
        B
        C
        D
        E
        F
        G
    end
```

This workflow is characterized by an huge amount of data to be processed, single-stage elaborations can be performed in **parallel**, mainly data-intensive based on **matrix elaborations** and other arithmetic operations, **few branch instructions and data dependencies**.

At the end the 3 main CG operations parallelized on GPU are:

1. Vertex Processing
2. Rasterization
3. Fragment/Pixel Processing

OpenGL/Direct3D workflow:

- Application provides shader program binaries to the GPU
- Application sets graphics pipeline parameters.
- Application provides to the GPU a buffer of vertices
- Application send to the GPU a drawing command

2.1.2 GPU for general-purpose computing

At some point there was there was the necessity of hardware accelerators designed to accelerate the graphics pipeline. To speed up we can divide the starting of the evolution of graphics pipeline accelerators into:

- Pre-GPU era: graphics supercomputers in the 70s/90s mainly targeted **specific market sections** such as computer simulation, digital content creation, engineering, and research.

- At the beginning **Fixed-function GPU:** GPUs with fixed-functionality for specific tasks, such as 3dfx voodoo (1996) and NVIDIA GeForce 256 (1999)
- Later **Programmable GPU** arrived: GPUs with programmable stages, such as NVIDIA GeForce 3 (2001), GeForce 6 (2004), and GeForce 7800 (2005)
- **GPU based on unified shader processors**: GPUs with unified shader processors, such as NVIDIA GeForce 8 (2006) and Tesla (2006) The 3 key ideas:
  - Instantiate many shader processors
  - Replicate ALU inside the shader processor to enable SIMD processing
  - Interleave the execution of many groups of SIMD threads

2.2 Basic architecture of GPU

The main idea is to use of many "slimmed down" and simplified cores to run in parallel. Simplified means:

- removed all components that help a single instruction stream run faster:
  - out of order logic out
  - branch prediction out
  - memory pre-fetcher out
  - and stuff like this, removed

At the end we obtain:

- A multicore of simplified processors: pack cores full of ALUs
- Sharing instruction streams across group of data chucks
- Implicit SIMD execution managed by HW

But because of the simplification we will have these problems:

- Branches impact on GPU performance
- The impact of stalls caused by data dependencies (for example memory access is 100x slower than an ALU instruction generally)

We move from a latency-oriented architecture (CPU) to a throughput-oriented one (GPU) to improve overall task execution latency:

- Latency of the single stream is **increased** (remember that all the components regarding the single instruction stream efficiency has been removed)
- But overall **throughput is decreased**
- So at the end, indirectly the latency of the **overall** task execution is decreased

To make this happen we have absolutely to write arithmetic intensive (high math/memory instructions ratio) tjat re-uses as much as possible the data fetched from global memory.

In general the architecture will:

- avoid latency stalls by interleaving execution of many groups of instruction streams: when a group stalls, work on another group
- the larger the arithmetic/memory instruction ratio, the lower the number of streams to hide memory stalls

and let's clarify that **coherent execution** (when the same instruction sequence applies to many data elements) is:

- necessary for SIMD processing resources to be used efficiently
- not necessary for efficient parallelization across different cores: indeed different cores independently executes different instructions streams

Divergent execution is a lack of instruction stream coherence.

At the end the takeaway is that the partition the data into groups and execute them concurrently on different groups of threads. This approach keeps the streaming multi-core busy by swapping to the next group of threads during memory latency periods.

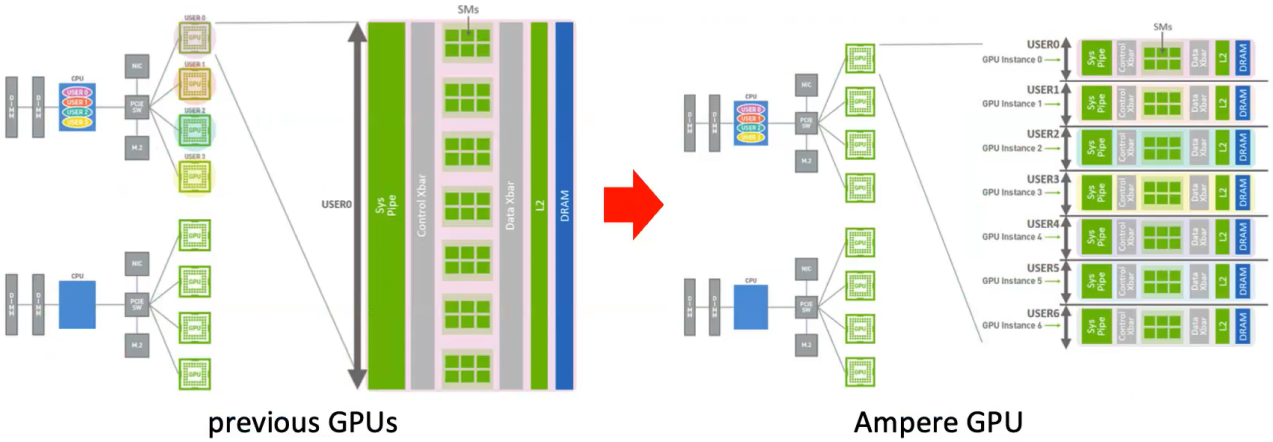
The idea is that each thread gets a unique index to determine its data portion.

Overall, our focus we be always to maximize data parallelism and throughput, writing a program featuring a high memory instruction ratio for automatic intensity. The shared memory, directly connected to the streaming multi-core, is a high-speed solution. However, the programmer is responsible for loading and transferring data to and from the shared memory. Profiling and optimization are essential to maximize performance.

2.2.1 Brief history of Nvidia GPUs

- **1999 - 2008**:
  - NVIDIA GeForce 256
    - First GPU, integrated transform and lighting (T&L) engine.
  - 2000: NVIDIA GeForce2
    - support for dual-texturing, and multi-texture blending.
  - 2001: NVIDIA GeForce3
    - Introduced programmable pixel and vertex shaders, support for DirectX 8.
  - 2002: NVIDIA GeForce4
    - Enhanced performance, introduced nView multi-display technology.
  - 2003: NVIDIA GeForce FX
    - Full support for DirectX 9, improved shader model capabilities.
  - 2004: NVIDIA GeForce 6 Series
    - Support for Shader Model 3.0, SLI (Scalable Link Interface) for multi-GPU setups.
  - 2005: NVIDIA GeForce 7 Series
    - Improved performance, introduced Transparency Anti-aliasing and PureVideo HD.
  - 2006: NVIDIA GeForce 8 Series
    - **CUDA** introduction
    - unified shader processor
  - 2008: NVIDIA GeForce 9 Series
    - Further performance improvements, support for Hybrid SLI.
  - 2008: NVIDIA GeForce GTX 200 Series
    - Introduced GTX branding, support for PhysX and CUDA improvements.
- **2010: Fermi** (Thread blocks, restructured caches)
  - Streaming Multiprocessor (SM) architecture introduced.
  - Programmer divides threads into blocks, which are dispatched by the global scheduler among SMs.
  - Each SM has 32 streaming cores (CUDA cores).
  - Threads inside a block are divided into groups of 32 called "warps," sharing an instruction stream.
  - SM has 2 scheduling and dispatching units, selecting two warps each clock cycle.
  - Register file supports up to 48 interleaved "warps."
  - restructured cache hierarchy including L1 and L2 caches which avoid necessity to access global memory
  - 6 64-bit memory partitions
- **2012: Kepler**
  - Dynamic parallelism: now a kernel can launch another kernel (GPU adapts to data, dynamically launches new threads)
  - New scheduler allows 2 independent instructions per warp per cycle, improving ILP (Instruction-Level Parallelism).
- **2014**: Maxwell
  - FP16 for AI and deep learning applications.
- **2016: GTX 10 Series, Pascal**
  - **Unified Memory**, transparent transmission of memory pages between CPU and GPU.

  - Automatic handling of page faults and global data coherence.
  - NVLink
- **2017: Volta
  - Tensor Cores introduced for matrix multiply-accumulate operations, benefiting deep neural networks.
  - Streaming multicore partitioned in processing blocks
- **2018: RTX 20 Turing**
  - RT unit for ray tracing additional CUDA cores
  - Acceleration for all data types: FP16, FP64, INT8, INT4, binary
- **2020: RTX 30 Ampere** (Virtualization capabilities)
  - Multi-instance GPU (MIG) virtualization and GPU partitioning feature in A100 supporting up to seven instances
  - Improved ray tracing and AI capabilities, GDDR6X memory, NVLink interface for multi-GPU architectures, support for all data types with new tensor cores.



- **2022: RTX 40 Ada Lovelace**
  - Further advancements in ray tracing, AI graphics with DLSS 3.0, improved power efficiency and performance


www.inve tec.com

2.2.2 Other GPU Vendors

AMD Timeline:

- **AMD Radeon HD 6970 Cayman (2010):**
  - The streaming multiprocessor is called a compute unit
  - threads are grouped in 64 elements (wavefront)
  - four clocks are used to execute an instruction for all fragments in a wavefront.
  - The single SIMD lane has a **VLIW** architecture
- **AMD Radeon R9 290X (2013):**
  - VLIW architecture replaced with a SIMD 16x vector architecture
  - Similar workflow of the NVIDIA counterpart (with wavefront (warp) interleaving)
  - The overall multicore architecture, including up to 44 compute units, and the memory hierarchy and communication infrastructure.

ARM Mali GPUs:

- **ARM Mali 628 (2014):**
  - Targeted for embedded computing with fewer shader cores compared to NVIDIA and AMD GPUs.
- **ARM Mali G720 (2023):**
  - Continued focus on embedded systems with improvements in performance and efficiency.

3 CUDA

CUDA (Compute Unified Device Architecture) is an extension of C++ (and also a lot of other languages like Fortran, Java, and Python) designed for writing programs that leverage GPUs for accelerating functions.

Introduced in 2006 with NVIDIA's Tesla architecture, CUDA's success relies on understanding GPU organization. CUDA is composed of software (language, API, and runtime), firmware (drivers and runtime), and hardware (CUDA-enabled GPU).

CUDA provides two main APIs for GPU device management:

- the CUDA Driver
- CUDA Runtime, which are mutually exclusive. On top of these, **CUDA includes libraries** for executing popular algorithms and functions, enhancing productivity and performance.

3.1 CUDA libraries

Examples of CUDA GPU-accelerated libraries to optimize performance and enhance software productivity are:

- **Math Libraries**: cuBLAS, cuFFT, CUDA Math Library, etc.
- **Parallel Algorithm Libraries**: Thrust
- **Image and Video Libraries**: nvJPEG, Video Codec SDK
- **Communication Libraries**: NVSHMEM, NCCL
- **Deep Learning Libraries**: cuDNN, TensorRT
- **Partner Libraries**: OpenCV, Ffmpeg, ArrayFire

Detailed list and more info: NVIDIA GPU-accelerated libraries

Common Library Workflow:

1. Create a library-specific handle
2. Allocate device memory for inputs/outputs.
3. Format inputs to library-specific formats.
4. Execute the library function.
5. Retrieve outputs and convert them if necessary.

6. Release CUDA resources and continue with the application.

4 Programming and execution models

The overview of an execution of a CUDA program is basically this:

1. **Data Transfer**: The main program runs on the CPU, loading data, transmitting it to the device.
2. **Kernel Execution**: The GPU program (kernel) is loaded and executed.
3. **Result Retrieval**: Computed results are transferred back from GPU to CPU memory.

Kernels execute across a set of parallel threads, which are organized into thread blocks. These blocks are organized into a grid that can be up to three-dimensional.

$$\text{kernel} \rightarrow \text{(2d or 3d) grid of blocks} \rightarrow \text{each blocks contains a (2d or 3d) grid of threads}$$

More specifically the execution model is:

1) Block
   - Blocks are assigned to Streaming Multiprocessor (SM)
   - Multiple blocks can be assigned to the same SM
   - Blocks are executed in any order, independent of each other
2) Warp
   - SM divides the block into warps (the number of warps depends on the number of threads per block)
   - Each warp is generally composed by 32 threads
   - SM interleaves the execution of several warps
   - While the programmer perceives the threads of a grid as concurrent at the application level, the actual execution order of blocks and warps is determined by the CUDA runtime and the GPU hardware at the architecture level.
3) SM Occupancy
   - High resource utilization is obtained by interleaving many warps in the SM
   - SM occupancy = `(Active Warps) / (Maximum Warps)`
   - Higher occupancy leads to better resource utilization: when a warp stalls, the SM executes another warp (latency hiding)

The SM divides the block in active warps (32 threads each) • A warp is executed with a SIMT approach • Execution of several warps is interleaved

Each active warp may be classified as:

- Selected warp if currently executed
- Stalled warp if not ready to be executed
- Eligible warp if ready to be executed

More specifically the CUDA latency hiding consists in having a large number of active threads (or warps) per streaming multiprocessor (SM) and interleaved them. While some threads are waiting for memory operations to complete (when a thread issues a memory request it may have to wait hundreds of clock cycles for the data to be fetched), other threads can be scheduled to perform **arithmetic operations**, effectively hiding the memory latency.

This ratio indicates how many arithmetic (compute) instructions are executed for every memory instruction

A higher ratio implies that the program does more computation relative to the amount of data it loads or stores, whereas a lower ratio suggests that the program is more memory-bound.

In summary, the larger the arithmetic/memory instruction ratio, the lower the number of streams needed to hide memory stalls. This is because a higher ratio implies more arithmetic work per memory operation, allowing each stream to effectively utilize the SM while waiting for memory operations to complete.

4.1 CUDA Thread Hierarchy

CUDA organizes threads hierarchically into grids, with each grid divided into blocks.

- Blocks are assigned to a streaming multiprocessor (SM) for execution
- Blocks can be executed in any order and are managed by SM
- The SM schedules execution of **warps**, which are groups of 32 threads. In any moment a warp inside a SM can be:
  - Selected warp: if currently executed
  - Stalled warp: if not ready to be executed
  - Eligible warp: if ready to be executed

In each warp, each thread is identified by a block number and thread number. In the code to correctly map each thread to a single data element and its corresponding input elements we have to create a flattened index for accessing the input and output arrays. We can use these variables:

- `blockIdx`: id of the block in the grid
- `blockDim`: size of the block (#threads)
- `threadIdx`: id of the thread in the block
- `gridDim`: size of the grid (#blocks)

The thread grid in CUDA can have up to three dimensions, selected based on the problem's nature:

- 1D problems involve vector operations
- 2D for matrices or images
- 3D for volumetric data

Actually even if the grid can be defined with up to 3 dimensions, the grid is internally linearized with a row major layout.

Warp divergence is one factor of performance degradation: It happens when threads in the same warp takes different directions during the execution of a branch or loop statement Strategies to minimize divergence include organizing data to ensure threads in the same warp perform same operations. The flow of CUDA calls involves enqueuing commands in a stream, with synchronization points used to manage execution order.

Function Qualifier	Executed on	Only Callable From
`__device__`	Device	Device
`__global__`	Host	Device
`__host__`	Host	Host

In CUDA programming, when transferring data from the host to the device for arithmetic instructions on the CPU side, we use the `cudaMemcpy()` function with the transmission direction set to "from host to device." The function is a **blocking** function, meaning control is returned to the main program once the transmission is completed. Typically, prefixes like `dev_` or `d_` are used to denote device (GPU) variables, but the important thing is to be consistent.

```
cudaMalloc (&d_va, N*sizeof (int));

// CPU->GPU data transmission
cudaMemcpy(d_va, h_va, N*sizeof (int), cudaMemcpyHostToDevice);
```

```
// kernel launch
dim3 blocksPerGrid (N/ 256, 1,1);
dim3 threadsPerBIock (256, 1,1);
vsumKerne1<<<blocksPerGrid, threadsPerBIock>>> (d_va, d_vb, d_vc);

// GPU->CPU data transmission * /
cudaMemcpy(h_vc, d_vc, N*sizeof (int), cudaMemcpyDeviceToHost) ;

// device memory freeing
cudaFree (d_va) ;
cudaFree (d_vb) ;
cudaFree (d_vc) ;
```

CUDA calls may be:

- Blocking: e.g `cudaMemcpy`
- Non-blocking: it continues asynchronously its execution e.g `kernel launch`

A barrier can be used to force the host to wait the termination of the kernel execution. Inside blocks, synchronization in CUDA is explicitly invoked by the programmer as the GPU is designed for high performance with no automatic synchronization mechanism. Logically, threads are concurrent, but the execution order is unpredictable. Synchronization is performed using barriers that synchronize threads in the same block. There is no barrier call applicable among threads of different blocks at grid level.

`__syncthreads()` is the barrier call.

5 CUDA memory model

The CUDA **memory model** is explicit to the programmer: caches are not entirely transparent, demanding a deeper understanding of memory hierarchy and access patterns. The device memories are mainly:

- **registers**: registers are fast, private, and limited
- **caches**: non-programmable memories transparent to the programmer:
  - L1 private to each streaming multi-processor
  - L2 private per SM
- **device memory**: (aka global or video memory) is slower and accessible by all threads and the host.

Also CUDA introduces the concept of **memory spaces** which help the programmer explicitly manage the placement and accessibility of variables. The three most important keywords are:

- `__global__`: Variables reside in global memory and can be accessed by all threads and the host.
- `__shared__`: Variables reside in shared memory and can be accessed by all threads within the same block.
- `__constant__`: Variables reside in constant memory and can be accessed by all threads and the host.

Variable name	Memory	Scope	Lifetime
int a	Register	Thread	Thread
int arr[10]	Local	Thread	Thread
shared int a or int arr[10]	Shared	Block	Block
device int a or int arr[10]	Global	Global	Application
constant int a or int arr[10]	Constant	Global	Application
restricted constant int *p	Texture/ Read-only	Global	Application

The local memory is used if the reserved number of registers is exceeded (register spilling) or for "large" variables like arrays or large structures.

Memory	On/Off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in a block	Block
Global	Off	Yes	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture/ Read-only	Off	Yes	R	All threads + host	Host allocation

5.1 CUDA memory functions

In addition to the memory types and spaces, CUDA also provides **memory management functions** for allocating, copying, and freeing memory on the device. These functions include:

- `cudaMalloc()`: Allocates memory on the device.
  - `addr`: Pointer to the allocated memory address on the device
  - `n`: Number of elements to allocate memory for

  `cudaMalloc(&addr, n * sizeof(int));`
- `cudaMemcpy()` -destPtr: Pointer to the destination memory address -srcPtr: Pointer to the source memory address -n: Number of elements to copy -cudaMemcpyHostToDevice: Copy data from host (CPU) to device (GPU) -cudaMemcpyDeviceToHost: Copy data from device (GPU) to host (CPU)

```
cudaMemcpy(destPtr, srcPtr, n * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(destPtr, srcPtr, n * sizeof(int), cudaMemcpyDeviceToHost);
```

- `kernelLaunch`
  - `kernel`: Name of the kernel function to launch
  - `blocksPerGrid`: Number of blocks per grid (can be 1D, 2D, or 3D)
  - `threadsPerBlock`: Number of threads per block
  - `arg1, arg2, ...`: Kernel function arguments

```
kernel<<<blocksPerGrid, threadsPerBlock>>>(arg1, arg2, ...);
```

- `cudaFree(ptr)`: Frees memory allocated on the device.
- `cudaMemcpy()`: Copies data between host and device memory, or between different memory spaces on the device.
- `cudaMemcpyAsync()`: Performs an asynchronous memory copy, allowing overlap of memory transfers with computation.
- `cudaMallocManaged()` to allocate memory that can be accessed from both the host and the device. With Unified Memory, explicit data transfers between host and device are no longer necessary.

5.2 Shared memory

Shared memory is a fast, on-chip memory that is shared among all threads within a block. It is a programmer-managed memory and can be used to efficiently share data between threads in the same block.

Shared memory is divided into equally-sized memory banks: it's crucial to avoid bank conflicts, which occur when multiple threads access different addresses within the same bank.

Shared memory supports broadcasting: in the case of a single address read by threads in the same block (1 transaction).

Proper usage of shared memory involves carefully designing access patterns to minimize bank conflicts and maximize memory bandwidth utilization.

An example of the use of shared memory is the **matrix-matrix Multiplication**:

- Compute each element of the output matrix as a product of a row from the first matrix and a column from the second matrix.
- Typically implemented using three nested for loops:
  - Two outer loops scan the entire output matrix.
  - One inner loop performs the dot product.
- Using Shared Memory:
  - Transfer the first strip of matrix M to shared memory once.
  - This allows multiple accesses by threads, which is more efficient than accessing global memory.
  - Shared memory can handle broadcast access better than global memory can handle transactions.
- Optimizing with Sub-Regions:
  - Although shared memory is limited, it can be divided into sub-regions for computation.
  - Computing the first sub-region against the second can optimize memory usage and computational efficiency.

```
graph TB
    A[Matrix Multiplication] --> B[Row from first matrix × Column from second matrix]
    B --> D[Three nested for loops]
    D --> E[Two outer loops: Scan output matrix]
    D --> F[Inner loop: Compute dot product]
    B --> G[Use shared memory]
    G --> H[Transfer first strip of matrix to shared memory]
    H --> I[Efficient multiple accesses by threads]
    I --> J[Divide shared memory into sub-regions]
    J --> K[Optimize memory usage and computation]
```

5.3 Constant memory

Constant memory is a read-only memory that is cached and optimized for broadcast access patterns, where all threads in a warp read the same memory location. It is ideal for storing constants or lookup tables that do not change during kernel execution.

Constant memory is limited in size (typically 64KB) and is accessible by all threads and the host. To use constant memory, variables must be declared with the `__constant__` qualifier and initialized on the host side using `cudaMemcpyToSymbol()`.

5.4 Texture memory

Texture memory is a read-only memory that is optimized for spatial locality and can be used to accelerate access to data with 2D or 3D locality.

Texture memory provides 2D caching, addressing, and filtering capabilities that can improve performance for certain access patterns. Textures are bound to a specific memory region and can be accessed using special texture fetching functions within the kernel.

Texture memory is particularly useful for image processing, computer vision, and applications that exhibit spatial locality in their data access patterns.

5.5 Global memory

Global memory usage in CUDA is used ensuring aligned memory accesses:

- GPUs read an entire line or sector when a single value is requested.
- In newer architectures, lines are divided into 32-byte sectors.
- Transactions must be aligned with the base address of the data (multiple of 32 bytes)
- Multiple accesses to the same cache line are combined into a single transaction.
- So, the optimal access strategy is to access 32 consecutive addresses (32-byte aligned) to minimize transactions.

When all accesses request values from the same cache line, efficiency is maximized. The worst-case scenario is when threads access random addresses in global memory: it may result in a separate transaction for each value.

5.5.1 Global memory efficiency

In general the load/store

$$\text{efficiency} = \frac{\text{(bytes loaded)}}{\text{(bytes used)}}$$

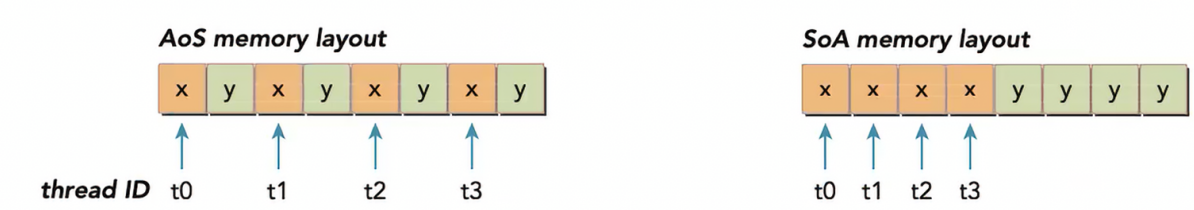
or more specifically:

$$\text{Global\_load\_efficiency} = \frac{\text{requested global memory load throughput}}{\text{required global memory load throughput}}$$

$$\text{Global\_store\_efficiency} = \frac{\text{requested global memory store throughput}}{\text{required global memory store throughput}}$$

5.5.2 Efficiency of Array of Structures vs. Struct of Arrays

Let's compare Array of Structures (AoS) with Structure of Arrays (SoA):




```
typedef struct {
    float x;
    float y;
} innerStruct_t;

innerStruct_t myAoS[N];
```

```
typedef struct {
    float x[N];
    float y[N];
} innerArray_t;
```

```
innerArray_t mySoA;
```

SoA in the example of RGB to grayscale conversion: data into separate arrays for red, green, and blue yields higher efficiency. SoA achieves 100% efficiency compared to only 33% with AoS.

The reason about that is because in SoA all the elements in the array are consecutive in memory.

Let's see another example:

- AoS:
  - `[x1 y1 z1] [x2 y2 z2] [x3 y3 z3]`
  - Threads accessing `x` fields would read non-contiguous memory locations.
  - This results in multiple memory transactions, reducing efficiency.
  - loading one field may bring unnecessary data into cache.
- SoA:
  - `[x1 x2 x3 ...] [y1 y2 y3 ...] [z1 z2 z3 ...]`
  - Threads accessing `x` fields read contiguous memory.
  - This can often be satisfied with a single memory transaction.

5.5.3 Instructions for synchronization

Atomic instructions are used to perform read-modify-write operations on global memory or shared memory, ensuring that no other thread can access the memory location simultaneously. These instructions are commonly used for synchronization and coordination among threads in a CUDA program.

OPERATION	FUNCTION	SUPPORTED TYPES
Addition	`atomicAdd`	int, unsigned int, unsigned long long int, float
Subtraction	`atomicSub`	int, unsigned int
Unconditional Swap	`atomicExch`	int, unsigned int, unsigned long long int, float
Minimum	`atomicMin`	int, unsigned int, unsigned long long int
Maximum	`atomicMax`	int, unsigned int, unsigned long long int
Increment	`atomicInc`	unsigned int
Decrement	`atomicDec`	unsigned int
Compare-And-Swap	`atomicCAS`	int, unsigned int, unsigned long long int
And	`atomicAnd`	int, unsigned int, unsigned long long int
Or	`atomicOr`	int, unsigned int, unsigned long long int
Xor	`atomicXor`	int, unsigned int, unsigned long long int

The **shuffle instruction** enables threads within the same warp to exchange data by directly accessing each other's registers without relying on shared or global memory. This allows for efficient data sharing and communication within a warp. The shuffle instruction is particularly useful when threads need to exchange data with other threads in the same warp without the overhead of using shared memory or global memory.

```
int __shfl_sync(unsigned mask, int var, int srcLane, int width=warpSize);
```

with:

- `mask`: A 32-bit integer representing the threads participating in the shuffle operation. Each bit corresponds to a thread in the warp, and only threads with their corresponding bit set will participate in the shuffle.
- `var`: The variable or register containing the data to be exchanged among threads.
- `srcLane`: The lane (thread) from which the data will be read. Each thread can specify the lane it wants to read data from.

- `width`: The number of threads participating in the shuffle operation. By default, it is set to the warp size (32).

Threads can perform data exchange, reduction operations, or other collaborative computations within a warp.

6 Task level parallelism streams, events, dynamic parallelism

In the first classes, we commented that CUDA supports two different types of parallelism: data parallelism and task parallelism.

During the first part of this course, we focused on data parallelism because the GPU is designed to exploit it through its parallel architecture. We dissected a single function to parallelize the execution on each data chunk or item.

Now let's talk about **task-level** parallelism: different tasks are performed simultaneously for more complex orchestrations of computations and data management.

6.1 Streams and Concurrency

CUDA introduces the concept of **streams** for task-level parallelism: a stream is a sequence of commands (like kernel executions or memory operations) that execute in order.

When multiple streams are used, the CUDA driver manages the execution order across different hardware queues. This was limited to a single queue before Fermi GPUs: creating potential false dependencies among tasks.

With the evolution of GPU architectures from Fermi to Kepler and beyond, NVIDIA introduced the ability to handle multiple hardware queues, thus enabling more sophisticated management of parallel tasks through the use of multiple streams.

Modern GPUs typically support 32 hardware streams. If the number of declared streams exceeds this limit, multiple software streams may map to the same hardware queue, potentially leading to contention and reduced performance.

In each stream, commands are pushed and popped using a first-in-first-out policy and forwarded to the GPU. On the GPU side, everything is managed directly in hardware, with commands executed sequentially per stream:

- Operations on the same stream: fifo, no overlap
- Operations on different streams : unordered and overlap
- PCIe executes a single transfer per direction:: multiple concurrent memory ops in same direction are serialized
- Parallelize data transfer and kernel execution on diff streams: split data and work in chunks.
- events can be used for synchronization

By using **streams**, programmers can effectively overlap data transfers and kernel executions, optimizing the utilization of both CPU and GPU resources. **Responsibility**: It's up to the programmer to design the application to take full advantage of task-level parallelism by carefully scheduling operations and managing dependencies.

CUDA provides two types of streams:

- the default stream
- non-default streams.

The default stream, or NULL stream, is automatically handled and requires no explicit intervention from the programmer.

To utilize non-default streams, you first declare and initialize them using CUDA API calls:

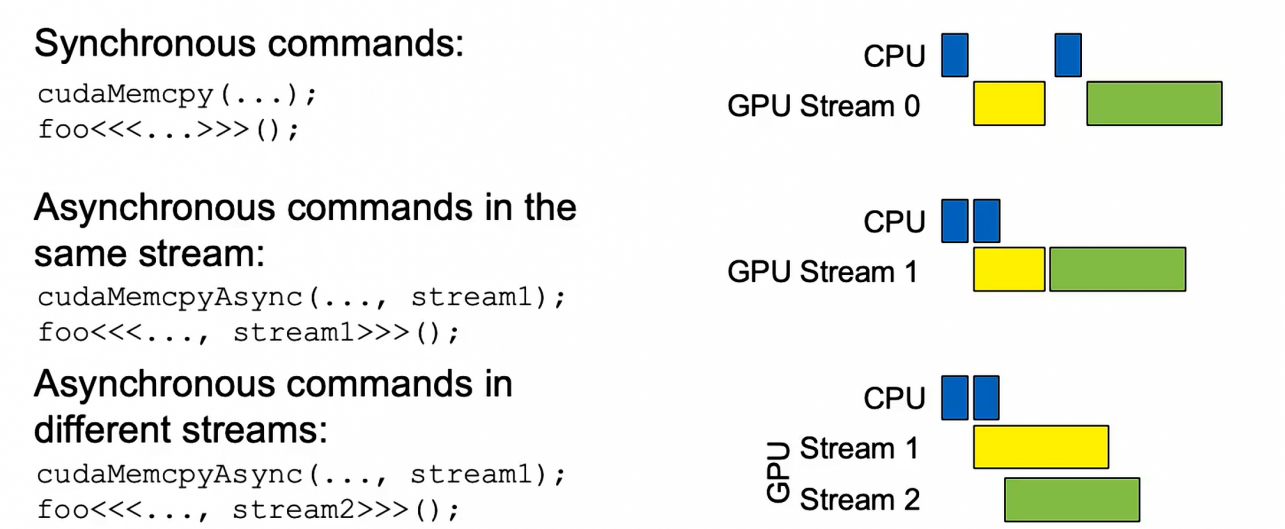
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
```

These streams allow for the parallel execution of tasks such as kernel execution and data transfers, enhancing the GPU's ability to perform multiple operations concurrently.

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(devPtr1, hostPtr1, size, cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(devPtr2, hostPtr2, size, cudaMemcpyHostToDevice, stream2);

kernel1<<<gridSize, blockSize, 0, stream1>>>(arguments1);
kernel2<<<gridSize, blockSize, 0, stream2>>>(arguments2);
```

`cudaMemcpyAsync` is the asynchronous counterpart of `cudaMemcpy`, pushing the data transfer command into a CUDA stream. This function allows the host thread to continue execution without waiting for the data transfer to complete.

```
cudaMemcpyAsync(destination, source, size, cudaMemcpyHostToDevice, stream);
```



6.1.1 Pinned Memory

Since the virtual memory mapping (introduced in Fermi architecture) and the rotation of pages by the OS can take to undefined behaviour, CUDA runtime uses pinned (locked) pages.

Pinned memory plays a crucial role in optimizing data transfers between the host and GPU, offering a significant performance boost by avoiding unnecessary copies and enabling asynchronous operations. When memory is allocated as "pinned" or "page-locked," it cannot be swapped out by the operating system, ensuring the memory address remains constant and accessible for DMA (Direct Memory Access) operations.

We can force a pinned page directly from the code using functions like `cudaMallocHost()` or `cudaHostAlloc()`, and freed with `cudaFreeHost()`. These functions are used instead of regular `malloc()` to allocate host memory that is page-locked and accessible to the device.

Pinned Memory Benefits:

- **Increased Data Transfer Performance**: Direct access by the DMA engine eliminates the need for copying data to a temporary pageable buffer before transferring to the GPU.
- **Enabling Asynchronous Data Transfer**: Asynchronous transfers allow the GPU to perform data transfer concurrently with other tasks, thereby maximizing the utilization of hardware resources.

```
cudaMallocHost(&ptr, size);  // Allocates pinned memory on the host
```

This replaces typical `malloc` or automatic/static array declarations with `cudaMallocHost`, which directly communicates with the OS to allocate memory in a pinned manner.

However, using pinned memory has its drawbacks:

- **Resource Constraints**: Pinned memory can limit the flexibility of the system's virtual memory management by restricting the OS from dynamically managing the memory space.
- **Potential Impact on Overall System Performance**: Overuse of pinned memory might degrade the performance of other applications or the system as a whole due to reduced memory flexibility.

6.2 Synchronization Techniques

Synchronization is necessary to ensure that data dependencies are respected among various operations. CUDA provides several mechanisms to synchronize operations both within a single stream and across multiple streams.

6.2.1 Implicit Synchronization in CUDA Function Calls

Regarding synchronization between the host (CPU) and the device (GPU): certain CUDA function calls inherently synchronize the host with the device, meaning that the host execution will block until the specified CUDA operations are completed. The most common functions that implicitly **synchronize** host and device:

• Memory Allocation: - `cudaMallocHost()` - `cudaHostAlloc()` - `cudaMalloc()` • Memory Copy: - `cudaMemcpy()`(non-async versions) • Memory Setting: - `cudaMemset()` (non-async versions) • Device Configuration: - `cudaDeviceSetCacheConfig()`

These functions block the host until the GPU operation is complete, ensuring implicit synchronization can be helpful, obviously locking the host can prevent it from performing other tasks that could otherwise overlap with GPU operations, reducing the overall efficiency of the application.

To minimize the performance impact of these synchronization points, consider the following strategies:

- **Asynchronous Operations**: Where possible, use asynchronous versions of memory operations (`cudaMemcpyAsync()`, `cudaMemsetAsync()`) along with CUDA streams to allow overlap of memory transfers with computation, both on the host and device.
- **Stream Synchronization**: Use `cudaStreamSynchronize()` to synchronize only specific streams rather than blocking the entire host, allowing other operations to continue concurrently.

6.2.2 Events for synchronization

CUDA events are markers that can be placed in streams to record execution status. They are particularly useful for profiling and for coordinating between the host and device:

- `cudaEvent_t event` is marker that can be pushed in the stream
  - 'cudaEventCreate(&event);
  - `cudaEventDestroy(event);`
- Device records a timestamp when it reaches the event
  - `cudaEventRecord(event, stream1);`
  - `cudaStreamWaitEvent(stream2, event, 0);`

```
cudaEvent_t event;
cudaEventCreate(&event);
cudaMemcpyAsync(devPtr, hostPtr, size, cudaMemcpyHostToDevice, stream1);
cudaEventRecord(event, stream1);
cudaStreamWaitEvent(stream2, event, 0); // Make stream2 wait for event
cudaEventDestroy(event);
```

6.2.3 Global and fine-grained synchronization

As said before, CUDA provides two types of streams:

- the default stream
- non-default streams.

Operations in the default stream are executed only after all operations in other streams are complete, and vice versa. This ensures a level of synchronization without explicit barriers but can limit concurrency.

To avoid blocking behavior typical of the default stream, use the `cudaStreamCreateWithFlags` function to create non-blocking streams:

```
cudaStream_t stream3;
cudaStreamCreateWithFlags(&stream3, cudaStreamNonBlocking);
```

Non-blocking streams allow operations within the stream to proceed without having to wait for operations in the default stream to complete, further enhancing concurrency.

While to ensure that all commands in a stream complete before the CPU proceeds we can use:

```
cudaStreamSynchronize(stream1);
```

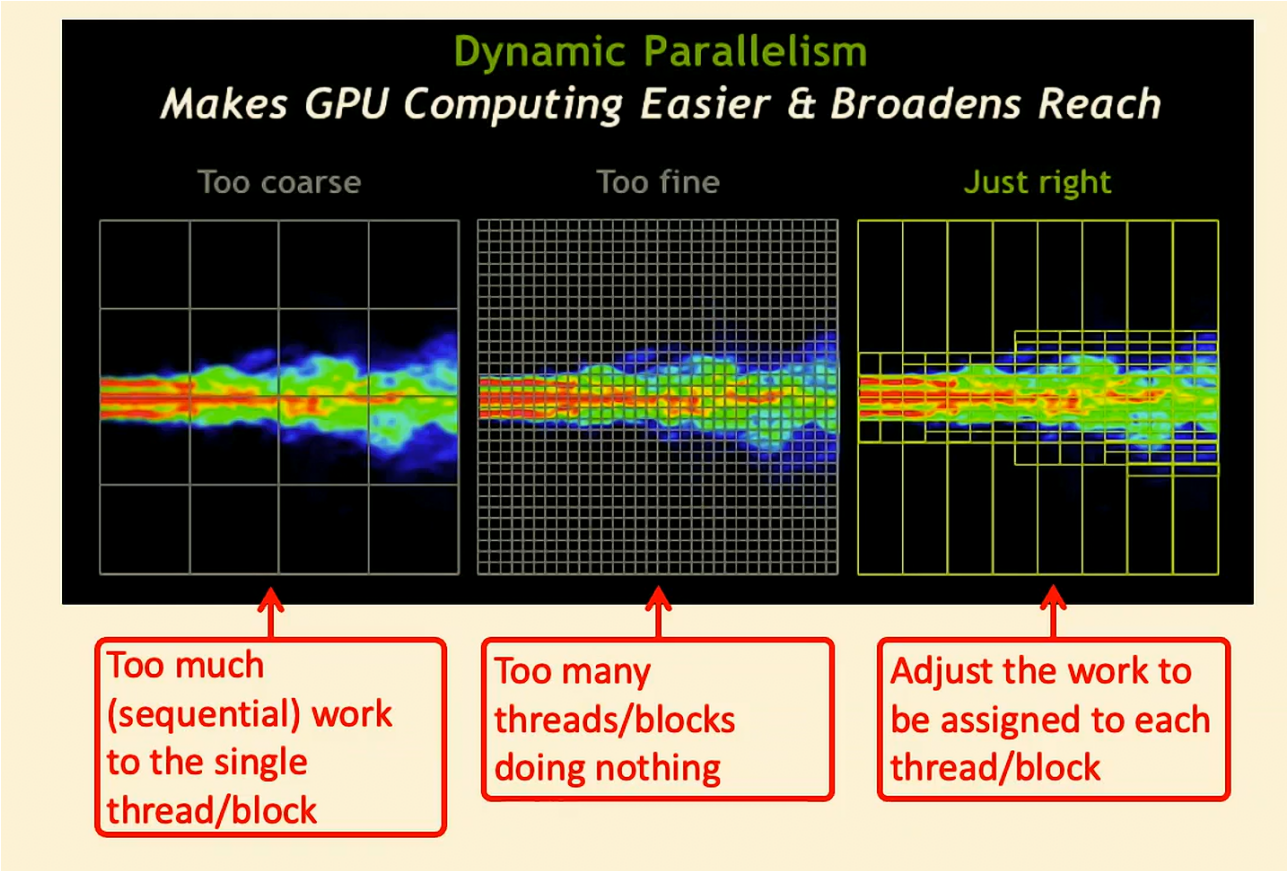
This places a barrier, ensuring the CPU only continues once all commands in `stream1` are complete.

`cudaDeviceSynchronize()`: Waits for the completion of all device activities and is considered a global barrier.

6.3 Dynamic parallelism

Dynamic parallelism (introduced in 2012 by Kepler) allows any thread within a kernel to invoke other instances of themselves to tackle more complex instance of the problem:

- **Adaptive Thread Management**: Useful for adjusting the number of threads for data sets with unpredictable sizes, like graphs or sparse matrices, which may vary during processing.
- **Reduced Host-Device Communication**: Eliminates the need for constant signaling between the host and device when additional tasks are found, allowing threads to directly launch new grids and bypassing the usual host intervention.



Specifically CUDA kernels can:

- Launch other kernels
- Allocate global memory
- Synchronize with child grids
- Create streams

However, the range of functions callable from within a device is more limited compared to those callable from the host.

- Max recursive depth is 24 levels theoretically, but practical limits due to GPU resource constraints are much lower.
  - The number of pending kernels is capped at 2048, and virtualization involves moving data to global memory, which can degrade performance.
- Each thread can independently launch a child kernel, which is asynchronous.
  - But to prevent launching a child grid per thread, only a specific thread (e.g., the first thread in a block or grid) should be programmed to launch the kernel to avoid creating excessive child grids.
- There's no predefined execution order between parent and child threads; the order is determined dynamically.
- cudaDeviceSync can be used to wait until all the child grids of current block ends.
- Implicit barrier in parent thread: child grid always terminates before parent thread / block / grid
- Child grids can access the parent grid's mobile, texture, and constant memory. However, simultaneous operations on the same data structure by both grids are not recommended due to the lack of guaranteed

execution order (weak consistency).

- By default, launching a kernel without specifying a stream uses the null stream, leading to serialized execution of child grids launched by multiple threads in the same block. To enable parallel execution, kernels should be launched in separate streams created with the CUDA stream non-blocking flag. Proper synchronization is necessary before stream destruction.

7 Tools CUDA compiler, profiler and debugger

Nvidia's documentation is comprehensive and well-maintained:

- Use the CUDA Programming Guide extensively for detailed information.
- For specific CUDA API references, consult the Runtime API Documentation.
- Access the full CUDA Programming Guide for an in-depth understanding.

CUDA API exposes both the Runtime and the Device API. - We will focus on the **CUDA Runtime API**: - The Runtime API simplifies device code management by providing implicit initialization, context management, and module management, leading to simpler code but less control than the Driver API. - The Device API allows for more fine-grained control, particularly over contexts and module loading, making kernel launches more complex. It's more bare metal.

The important difference is that you use the device API to build all the abstraction you will need. ### Best Practices when programming with CUDA

- Use the `__restrict__` keyword to suggest that pointers point to unique memory.
- If something is constant, use `const`.
- Prefer compile-time computations: utilize `macros` and `constexpr` for efficiency.
- Utilize `CHECK()` functions to verify execution status: but enable checks only in debug mode for better performance.
- `inline` is a keyword in C and C++ used to suggest that the compiler replace a function call with the function's actual code at the call site to potentially improve performance by reducing function call overhead. cautiously:
  - Typically used for small, frequently called functions where the overhead of a function call might be significant compared to the function's execution time. In CUDA programming, inline functions can be particularly useful for device functions that are called frequently within kernels, as they can help reduce register usage and improve performance.
  - Compilers treat it as a hint rather than a guarantee.
  - For force inlining, use `__attribute__((always_inline))`. "inline" Here's a summary of its key aspects:
- Be mindful of `#pragma unroll`:
  - It can boost performance but also increase register usage.
- Pay attention to synchronization functions:
  - Use `__syncthreads` or `__syncwarp` where necessary.
  - Fences (`__threadfence` and its variants) might also be options.

7.1 Compiling

CUDA programs are compiled using the `nvcc` compiler.

To compile: `nvcc vector_sum.cu -o vector_sum` To execute: `./vector_sum`

Compile for virtual and specific architectures using `nvcc` with flags like `-arch=compute_50` or `-arch=sm_50`.

Compilation Types: - **Just-in-Time (JIT):** PTX is included, runtime Cubin generation introduces overhead. - **Ahead-of-Time (AOT):** Cubin is directly in the executable, avoiding runtime compilation.

7.2 Debugging

- **Error Handling**: Every CUDA API function returns a flag which can be used for error handling. CUDA API calls return error codes that must be checked to ensure correct execution.
- **To measure kernel performance**: CPU timers or a GPU timer can be used to record the execution time of a kernel, including the launch time and execution time on the GPU.
- **Profiling and Tuning**: CUDA programs often require profiling and tuning to identify and address bottlenecks. This process involves iterative adjustments to grid and block sizes, memory access patterns, and other critical parameters, guided by profiling tools and best practices.

CUDA-GDB: Interactive debugger for CUDA code. - Requires `-g` and `-G` flags during compilation for debugging kernel code. - Supports breakpoints, switching execution context, and examining variables within kernels.

When you execute NVCC, your code is divided into device code and host code. The device code goes to the device compiler, and you get what's so-called Fat binary, of code FAT binary output all together, and you get your output file, which is the one you execute.

CUDA Binary Utilities

CUDA binary files are ELF-formatted and consist of executable code sections.

NVCC embeds cubin files into the host executable.

CUDA objdump, cuobjdump, nvdisasm: Tools for analyzing CUDA binaries, examining PTX or Cubin code for optimization purposes.

7.2.1 CUOBJDUMP & NVDISASM

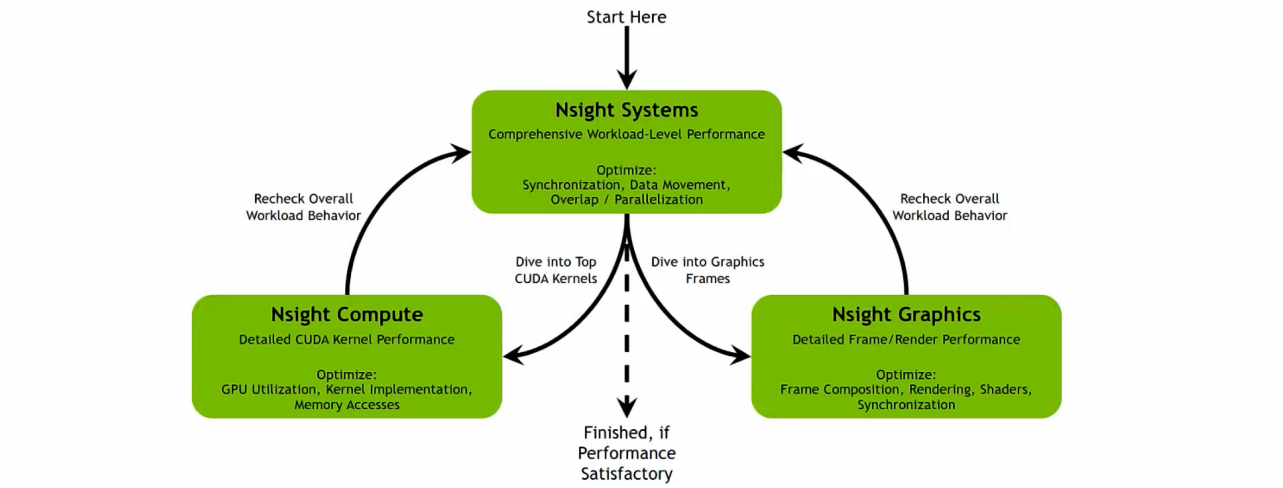
- CUOBJDUMP: Extracts and formats information from CUDA binary files.
- NVDISASM: Similar to CUOBJDUMP but for standalone cubin files.

7.3 Profiling and Metrics

The old way is: - NVPROF command-line tool for profiling data. - NVIDIA Visual Profiler (NVVP) for visualization and optimization.

The support ended with the Volta architecture. Now the **Profiling Tools** is **Nsight Profiling Tools (NCU):**

- **Nsight systems** for system-wide profiling.
- **Nsight compute** for interactive kernel profiling.

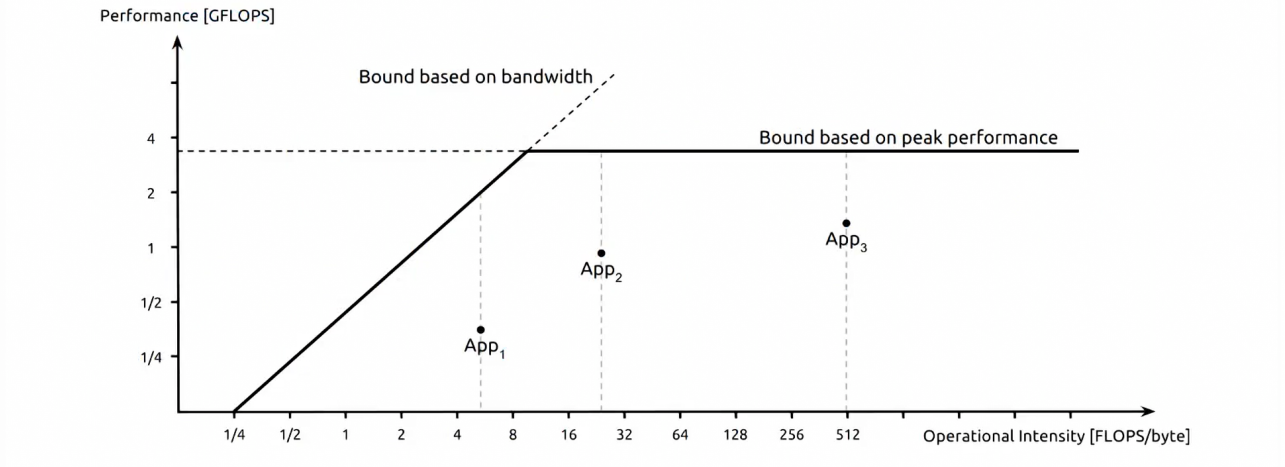


7.4 Roofline

The Roofline Performance Model is a visual tool for analysing performance bottlenecks and potential optimizations:

- **Computational Rooflines:** Represent limits based on operation type (floating-point, integer, fused multiply-add).
- **Memory Rooflines:** Indicate bandwidth limitations for different memory hierarchies (DRAM, L1/L2 caches, shared memory).

Example:



The roofline model offers a sophisticated visual framework: it shows performance boundaries by illustrating the balance between computational power and memory bandwidth.

It is based on the concept of Arithmetic (or operational) intensity:

$$\text{Arithmetic Intensity} = \frac{\text{Number of Floating Point Operations}}{\text{Bytes Transferred}} = [FLOPS/byte]$$

The roofline model provides performance estimates based on both computational throughput and bandwidth peak performance.

This graphical approach allows developers to pinpoint whether a program is compute-bound or memory-bound.

In practice, achieving optimal performance involves maximizing the operational intensity, thereby pushing the application towards the upper performance limits of the available hardware.

With Nsight Compute is possible to generate a roofline analysis for any given kernel. The aim is to move data as close to the computational core as possible, enhancing the use of faster cache layers to reduce the latency and bandwidth constraints imposed by slower global memory accesses.

Visually, the model shows **multiple rooflines** corresponding to different levels of memory hierarchy. This overlay provides a clear depiction of the achievable performance based on the arithmetic intensity of the application and the specific memory hierarchy being utilized.

8 CUDA Kernel optimizations

To maximize compute performance in CUDA kernels there are common patterns and “rules” that we can follow.

Optimization	Strategy	Benefit to Compute Cores	Benefit to Memory
Maximizing occupancy	Tune usage of SM resources (threads per block, shared memory per block, registers per thread)	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency
Enabling coalesced global memory accesses	Transfer between global and shared memory in a coalesced manner; Rearrange thread-to-data mapping and data layout	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache lines
Minimizing control divergence	Rearrange thread-to-work/data mapping and data layout	High SIMD efficiency (fewer idle cores during SIMD execution)	-
Tiling of reused data	Place reused data within a block in shared memory or registers	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic
Privatization	Apply partial updates to private data copies before updating the universal copy	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates
Thread coarsening	Assign multiple units of parallelism to each thread	Less redundant work, divergence, or synchronization	Less redundant global memory traffic

8.1 Maximizing Occupancy

The goal is to hide memory latencies by fully utilizing all available resources (threads and resources). This allows the GPU scheduler to continue executing work, effectively hiding the latency of memory operations from both compute and memory perspectives. A starting point to minimize kernel execution time is to maximize SM occupancy. When manually tuning the block thread size, consider the following guidelines:

- Ensure the block size is a multiple of the warp size.
- Avoid using small block sizes.
- Adjust the block size according to the kernel's resource requirements.
- The total number of blocks should significantly exceed the number of SMs.
- Engage in systematic experiments to find the best configuration.

8.2 Memory Access

Efficient memory access patterns are crucial for performance.

```
graph TD
    A[Is the usage pattern known, local, and not more than ~48KB?] -->|Yes| B[Shared memory!]
    A -->|No| C[Is read-only memory OK?]
    C -->|Yes| D[Is it less than ~64KB?]
    C -->|No| E[Hope you've been nice to your L2 cache...]
    D -->|Yes| F[Constant memory!]
    D -->|No| G[Texture memory!]
```

Coalesced memory access refers to the pattern of global memory access where threads in a warp access contiguous memory addresses.

- Applies to **global memory** access
- Aims to **minimize the number of memory transactions**
- Improves global memory bandwidth utilization

Shared memory can be used to avoid **un-coalesced memory** accesses by loading and storing data in a **coalesced pattern** from global memory and then reordering it in shared memory.

Shared memory can *only* be loaded by memory operations performed by threads in CUDA kernels. There are no API functions to load shared memory.

If you have an array that is larger than the number of threads in a threadblock, you can use a looping approach like this:

```
#define SSIZE 2592

__shared__ float TMshared[SSIZE];

  int lidx = threadIdx.x;
  while (lidx < SSIZE){
    TMShared[lidx] = TM[lidx];
    lidx += blockDim.x;}

__syncthreads();
```

In a naive **SGEMM** (simple general matrix multiply) kernel, memory access patterns can lead to non-coalesced access:

```
// non-coalesced memory access
__global__ void sgemm_naive(int M, int N, const float *A, const float *B, float beta, float *C) {
   const uint x = blockIdx.x * blockDim.x + threadIdx.x;
   const uint y = blockIdx.y * blockDim.y + threadIdx.y;
   // ... rest of the kernel ...
}
```

To ensure consecutive threads access consecutive memory locations:

```
// Example of a coalesced memory access pattern
__global__ void sgemm_coalesced(int M, int N, const float *A, const float *B, float *C) {
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;
 if (Row < M && Col < N) {
     float Cvalue = 0.0;
     for (int k = 0; k < K; ++k) {
```

```
            Cvalue += A[Row * K + k] * B[k * N + Col];
        }
        C[Row * N + Col] = Cvalue;
    }
}
```

8.2.1 Bank Conflicts ‘

As said before, shared memory is useful to increase the kernel performance, but **memory bank conflicts** can happen if not correctly managed. Bank conflicts occur when multiple threads in a warp access different words in the same memory bank of shared memory.

- Primarily a concern for **shared memory** access
- Aims to prevent serialization of memory accesses within a warp
- Improves shared memory access efficiency

A basic matrix transpose without using shared memory:

```
__global__ void transposeNaive(float * __restrict__ odata, const float * __restrict__ idata) {
  const int x = blockIdx.x * TILE_DIM + threadIdx.x;
  const int y = blockIdx.y * TILE_DIM + threadIdx.y;
  const int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

A naive solution using shared memory which **gives bank collisions** :

```
__global__ void transposeCoalesced(float *odata, const float *idata) {
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

During `tile[threadIdx.y+j][threadIdx.x]` something like this can happen:

(0,0) Bank 0	(1,0) Bank 1	(2,0) Bank 2	(3,0) Bank 3	(4,0) Bank 4	(5,0) Bank 5	(6,0) Bank 6	(7,0) Bank 7
(0,1) Bank 0	(1,1) Bank 1	(2,1) Bank 2	(3,1) Bank 3	(4,1) Bank 4	(5,1) Bank 5	(6,1) Bank 6	(7,1) Bank 7
(0,2) Bank 0	(1,2) Bank 1	(2,2) Bank 2	(3,2) Bank 3	(4,2) Bank 4	(5,2) Bank 5	(6,2) Bank 6	(7,2) Bank 7
(0,3) Bank 0	(1,3) Bank 1	(2,3) Bank 2	(3,3) Bank 3	(4,3) Bank 4	(5,3) Bank 5	(6,3) Bank 6	(7,3) Bank 7
(0,4) Bank 0	(1,4) Bank 1	(2,4) Bank 2	(3,4) Bank 3	(4,4) Bank 4	(5,4) Bank 5	(6,4) Bank 6	(7,4) Bank 7
(0,5) Bank 0	(1,5) Bank 1	(2,5) Bank 2	(3,5) Bank 3	(4,5) Bank 4	(5,5) Bank 5	(6,5) Bank 6	(7,5) Bank 7
(0,6) Bank 0	(1,6) Bank 1	(2,6) Bank 2	(3,6) Bank 3	(4,6) Bank 4	(5,6) Bank 5	(6,6) Bank 6	(7,6) Bank 7
(0,7) Bank 0	(1,7) Bank 1	(2,7) Bank 2	(3,7) Bank 3	(4,7) Bank 4	(5,7) Bank 5	(6,7) Bank 6	(7,7) Bank 7

To eliminate shared memory bank conflicts `__shared__ float tile[TILE_DIM][TILE_DIM+1]` add padding is enough:

(0,0) Bank 0	(1,0) Bank 1	(2,0) Bank 2	(3,0) Bank 3	(4,0) Bank 4	(5,0) Bank 5	(6,0) Bank 6	(7,0) Bank 7	PAD Bank 0
(0,1) Bank 1	(1,1) Bank 2	(2,1) Bank 3	(3,1) Bank 4	(4,1) Bank 5	(5,1) Bank 6	(6,1) Bank 7	(7,1) Bank 0	PAD Bank 1
(0,2) Bank 2	(1,2) Bank 3	(2,2) Bank 4	(3,2) Bank 5	(4,2) Bank 6	(5,2) Bank 7	(6,2) Bank 0	(7,2) Bank 1	PAD Bank 2
(0,3) Bank 3	(1,3) Bank 4	(2,3) Bank 5	(3,3) Bank 6	(4,3) Bank 7	(5,3) Bank 0	(6,3) Bank 1	(7,3) Bank 2	PAD Bank 3
(0,4) Bank 4	(1,4) Bank 5	(2,4) Bank 6	(3,4) Bank 7	(4,4) Bank 0	(5,4) Bank 1	(6,4) Bank 2	(7,4) Bank 3	PAD Bank 4
(0,5) Bank 5	(1,5) Bank 6	(2,5) Bank 7	(3,5) Bank 0	(4,5) Bank 1	(5,5) Bank 2	(6,5) Bank 3	(7,5) Bank 4	PAD Bank 5
(0,6) Bank 6	(1,6) Bank 7	(2,6) Bank 0	(3,6) Bank 1	(4,6) Bank 2	(5,6) Bank 3	(6,6) Bank 4	(7,6) Bank 5	PAD Bank 6
(0,7) Bank 7	(1,7) Bank 0	(2,7) Bank 1	(3,7) Bank 2	(4,7) Bank 3	(5,7) Bank 4	(6,7) Bank 5	(7,7) Bank 6	PAD Bank 7

```
// Avoiding bank conflicts in shared memory
__shared__ float shared_data[32][33];  // Padding to avoid conflicts
int tid = threadIdx.x;
shared_data[tid][threadIdx.y] = some_value;
```

```
__global__ void transposeNoBankConflicts(float *odata, const float *idata) {
  __shared__ float tile[TILE_DIM][TILE_DIM+1]; //this +1 changed

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

8.3 Privatization

Privatization consists in doing partial updates to private data copies before updating the universal copy. This reduces the need for resources:

- minimizes contention and serialization of atomic operations.
- Which means less pipeline stalls

8.4 Minimizing control divergence

This strategy involves rearranging thread-to-work/data mapping and data layout to minimize situations where threads in the same warp take different execution paths.

In general we want to minimize divergent branching within a warp to avoid serialization

Now, let's say that we have 4 threads in a warp (in reality, it's 32, but 4 is easier to visualize) and a 1D array of data, and we want to compute a histogram. We'll represent it like this:

```
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O][P]
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Without a stride-based approach, the threads might work like this:

```
Thread 0: [A][B][C][D]
Thread 1: [E][F][G][H]
Thread 2: [I][J][K][L]
Thread 3: [M][N][O][P]
```

This is a mis-aligned memory access but it has also consequences on control divergence. With a stride-based approach:

1. All threads are processing elements at the same stride (distance) from each other.
2. In each iteration, threads are more likely to be processing similar types of data (e.g., all at the beginning, all in the middle, etc.).
3. If there's any pattern in the data, all threads will encounter this pattern together.
4. Even if divergence occurs, it's more likely to affect all threads similarly, rather than causing some threads to do much more work than others.

```
Iteration 1:
    Thread 0: [A][ ][ ][ ]
    Thread 1: [B][ ][ ][ ]
    Thread 2: [C][ ][ ][ ]
    Thread 3: [D][ ][ ][ ]

Iteration 2:
    Thread 0: [A][E][ ][ ]
    Thread 1: [B][F][ ][ ]
    Thread 2: [C][G][ ][ ]
    Thread 3: [D][H][ ][ ]

Iteration 3:
    Thread 0: [A][E][I][ ]
    Thread 1: [B][F][J][ ]
    Thread 2: [C][G][K][ ]
    Thread 3: [D][H][L][ ]

Iteration 4:
```

```
Thread 0: [A][E][I][M]
Thread 1: [B][F][J][N]
Thread 2: [C][G][K][O]
Thread 3: [D][H][L][P]
```

8.5 Tiling of reused data

Tiling involves dividing a large data set into smaller chunks, or tiles, and processing them in parallel. This strategy involves placing reused data within a block in shared memory or registers. It reduces pipeline stalls waiting for global memory accesses and decreases global memory traffic.

The general outline of tiling technique:

1) Identify a tile of global memory content that are accessed by multiple threads
2) The tile from global memory into on-chip memory
3) Have the multiple threads to access their data from the on-chip memory
4) Move on to the next tile

8.6 Thread Coarsening

Coarsening refers to increasing the amount of work done by each thread in a kernel. > "Instead of having many threads doing small amounts of work, fewer threads do more work each."

This could seem counterintuitive, but for instance in the context of processing an image, instead of assigning one thread per pixel, thread coarsening would assign one thread to handle a block of pixels, reducing the total number of threads and potentially making better use of the GPU's resources.

Thread Coarsening:

- reduces redundant work, divergence, or synchronization
- decreases redundant global memory traffic

9 Histogram

A histogram is a display of the number count of occurrences of data values in a dataset. Often the data items are grouped into specific ranges or bins (e.g. a specific color in an image, a letter (or group of letters) in a text). They are used whenever there is a large volume of data that needs to be analyzed to distill interesting events (think of feature extraction in computer vision). Parallelization Strategy:

- Each thread processes a portion of the input array.
- Atomic operations are used to update the global histogram to avoid race conditions.

```
__global__ void histogram_kernel(const char *__restrict__ data,
                                 unsigned int *__restrict__ histogram,
                                 const unsigned int length) {
const int i = threadIdx.x + blockIdx.x * blockDim.x;

if (i < length) {
    const int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE)
      atomicAdd(&(histogram[alphabet_position / CHAR_PER_BIN]), 1);
}
```

9.1 Optimizations

9.1.1 Coarsening

Each thread processes multiple elements.

```
__global__ void
    histogram_kernel(const char *__restrict__ data, unsigned int *__restrict__ histogram, const int lengt
  const unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const unsigned int stride = blockDim.x * gridDim.x;

  // All threads in a blockk handle consecutive elements in each iteration
  for (unsigned int i = tid; i < length; i += stride) {
    const int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE)
      atomicAdd(&(histogram[alphabet_position / CHAR_PER_BIN]), 1);
  }
}
```

9.1.2 Privatization

Each thread maintains a local histogram before merging into the global one. We can privatize at different levels:

- shared memory
- registers
- Or even committing on different region of global memory to boost performance.

```
__global__ void histogram_kernel(const char *__restrict__ data,
                                 unsigned int *__restrict__ histogram,
                                 const unsigned int length) {

  const unsigned int tid    = blockIdx.x * blockDim.x + threadIdx.x;
  const unsigned int stride = blockDim.x * gridDim.x;
  // Privatized bins
  __shared__ unsigned int histo_s[BIN_NUM];
#pragma unroll
  for (unsigned int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) { histo_s[binIdx] = 0;
  __syncthreads();
  // Histogram
  for (unsigned int i = tid; i < length; i += stride) {
    const int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE)
      atomicAdd(&(histo_s[alphabet_position / CHAR_PER_BIN]), 1);
  }
  __syncthreads();
// Commit to global memory
#pragma unroll
  for (unsigned int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) {
    const unsigned int binValue = histo_s[binIdx];
    if (binValue > 0) {
      atomicAdd(&(histogram[binIdx]), binValue);
    }
  }
}
```

This approach uses a private histogram in shared memory before committing to the global histogram, reducing contention.

```
__global__ void histogram_kernel(const char *__restrict__ data,
                                 unsigned int *__restrict__ histogram,
                                 const unsigned int length) {
  const unsigned int tid    = blockIdx.x * blockDim.x + threadIdx.x;
  const unsigned int stride = blockDim.x * gridDim.x;

  // Privatized bins
  unsigned int histo_p[BIN_NUM];

__shared__ unsigned int histo_s[BIN_NUM];
#pragma unroll
for (unsigned int i = threadIdx.x; i < BIN_NUM; i += blockDim.x) {
    histo_s[i] = 0;
}
__syncthreads();

#pragma unroll
  for (unsigned int i = 0; i < BIN_NUM; i++) histo_p[i] = 0u;
  // Histogram
  for (unsigned int i = tid; i < length; i += stride) {
    const int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE)
      histo_p[alphabet_position / CHAR_PER_BIN] += 1;
  }
  // Commit to shared memory
#pragma unroll
  for (unsigned int binIdx = 0; binIdx < BIN_NUM; binIdx++) {
    const unsigned int binValue = histo_p[binIdx];
    if (binValue > 0) {
      atomicAdd(&(histo_s[binIdx]), binValue);
    }
  }

__syncthreads(); // Synchronization barrier

// Commit to global memory
#pragma unroll
  for (unsigned int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) {
    const unsigned int binValue = histo_s[binIdx];
    if (binValue > 0) {
      atomicAdd(&(histogram[binIdx]), binValue);
    }
  }
}
```

9.1.3 Aggregation

The concept of aggregating data values in contiguous regions to reduce atomic operations can be beneficial. To reduce high contention consecutive updates to the same bin are combined.

```
__global__ void histogram_kernel(const char *__restrict__ data,
                                 unsigned int *__restrict__ histogram,
                                 const unsigned int length) {
  const unsigned int tid    = blockIdx.x * blockDim.x + threadIdx.x;
  const unsigned int stride = blockDim.x * gridDim.x;
  // Privatized bins
  __shared__ unsigned int histo_s[BIN_NUM];
#pragma unroll
  for (unsigned int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) { histo_s[binIdx] = 0u;
  __syncthreads();

  // Histogram
  unsigned int accumulator = 0;
  int prevBinIdx           = -1;
  for (unsigned int i = tid; i < length; i += stride) {
    int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE) {
      const int bin = alphabet_position / CHAR_PER_BIN;
      if (bin == prevBinIdx) {
        ++accumulator;
      } else {
        if (accumulator > 0) {
          atomicAdd(&(histo_s[prevBinIdx]), accumulator);
        }
        accumulator = 1;
        prevBinIdx  = bin;
      }
    }
  }
  if (accumulator > 0) {
    atomicAdd(&(histo_s[prevBinIdx]), accumulator);
  }
  __syncthreads();
// Commit to global memory
#pragma unroll
  for (unsigned int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) {
    const unsigned int binValue = histo_s[binIdx];
    if (binValue > 0) {
      atomicAdd(&(histogram[binIdx]), binValue);
    }
  }
}
```

10 Convolution

Convolution is a mathematical operation that combines two functions to produce a third function.

Apart from image processing (blurring, sharpening, embossing, edge detection), the convolution operation is the foundation of Convolutional Neural Networks.

It involves sliding a filter (or kernel) over an input, performing element-wise multiplication and summation at each position, resulting in a transformed output that emphasizes or detects specific features in the data.

If we assume that the dimension of the filter is $(2r_x + 1)$ in the x dimension and $(2r_y + 1)$ in the y dimension, the calculation of each P element can be expressed as follows:

$$P(i, j) = \sum_{m=-r_y}^{r_y} \sum_{n=-r_x}^{r_x} I(i + m, j + n) \cdot F(m, n)$$

where:

$P(i, j)$ is the output pixel at position (i, j) $I(i + m, j + n)$ is the input pixel at position $(i + m, j + n)$ $F(m, n)$ is the filter coefficient at position (m, n)
Here's a basic CPU implementation of convolution:

```
void convolution_cpu(input_type *input, const input_type *filter, input_type *output,
                     int width, int height, int filter_size, int filter_radius) {
    // Iterate over each output pixel
    for (int outRow = 0; outRow < height; outRow++) {
        for (int outCol = 0; outCol < width; outCol++) {
            input_type value = 0.0f;

            // Apply the filter
            for (int row = 0; row < filter_size; row++) {
                for (int col = 0; col < filter_size; col++) {
                    int inRow = outRow - filter_radius + row;
                    int inCol = outCol - filter_radius + col;

                    if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                        value += filter[row * filter_size + col] *
                                 input[inRow * width + inCol];
                    }
                }
            }
            output[outRow * width + outCol] = value;
        }
    }
}
```

10.0.1 Constant memory

The convolution kernel benefits from using **constant memory**, which is optimized for situations where all threads read the same value, such as filter coefficients in convolution operations.

- **Constant Memory Benefits:**
  - A constant memory cannot be modified by threads during kernel execution
  - The size of the constant memory is quite small (64KB) and efficient
  - It is a read only cache
  - Less complex hardware to manage it
  - good performance only when each thread accesses the same data
  - Serial access to the same location is fast.
  - It's initialized once and accessible across multiple kernel launches.
  - Declared at the global level with `__constant__`.

```
#define FILTER_RADIUS 4
#define FILTER_SIZE   (FILTER_RADIUS * 2 + 1)
```

```
__constant__ filter_type constant_filter[FILTER_SIZE][FILTER_SIZE];

// Copy filter to constant memory
cudaMemcpyToSymbol(constant_filter, filter, FILTER_SIZE * FILTER_SIZE * sizeof(filter_type));
```

- **Symbolic Filter:** Treating the filter as a symbol simplifies the kernel call.
- **Loop Unrolling:** Fixed filter size allows compiler optimizations.

Here's a GPU kernel using constant memory:

```
__global__ void convolution_constant_mem_kernel(const input_type *__restrict__ input, input_type *__restr

    const int outCol = blockIdx.x * blockDim.x + threadIdx.x;
    const int outRow = blockIdx.y * blockDim.y + threadIdx.y;
    input_type value{0.0f};

    #pragma unroll
    for (int row = 0; row < FILTER_SIZE; row++) {
        #pragma unroll
        for (int col = 0; col < FILTER_SIZE; col++) {
            const int inRow = outRow - FILTER_RADIUS + row;
            const int inCol = outCol - FILTER_RADIUS + col;
            if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                value += constant_filter[row][col] * input[inRow * width + inCol];
            }
        }
    }
    output[outRow * width + outCol] = value;
}
```

10.0.2 Coarsening

Coarsening involves each thread performing more work, which can improve performance:

1. Increased arithmetic intensity
2. Reduced launch overhead
3. Improved memory access patterns
4. Better resource utilization

In the context of convolution operations, coarsening can be applied by having each thread compute multiple output elements instead of one.

```
__global__ void convolution_constant_mem_coarsening_kernel(const input_type *__restrict__ input,
                                                            input_type *__restrict__ output,
                                                            const int width, const int height) {
    const int outCol = blockIdx.x * blockDim.x + threadIdx.x;
    const int outRow = blockIdx.y * blockDim.y + threadIdx.y;
    const int stride_col = gridDim.x * blockDim.x;
    const int stride_row = gridDim.y * blockDim.y;

    for (int c = outCol; c < width; c += stride_col) {
        for (int r = outRow; r < height; r += stride_row) {
            input_type value = 0.0f;
            #pragma unroll
            for (int row = 0; row < FILTER_SIZE; row++) {
```

```
                #pragma unroll
                for (int col = 0; col < FILTER_SIZE; col++) {
                    const int inRow = r - FILTER_RADIUS + row;
                    const int inCol = c - FILTER_RADIUS + col;
                    if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                        value += constant_filter[row][col] * input[inRow * width + inCol];
                    }
                }
            }
            output[r * width + c] = value;
        }
    }
}
```

10.0.3 Shared Memory Optimization

The Arithmetic Intensity analysis becomes more complex:

$$\text{OUT\_TILE\_DIM}^2 \cdot (2 \cdot \text{FILTER\_RADIUS} + 1)^2 \cdot 2$$

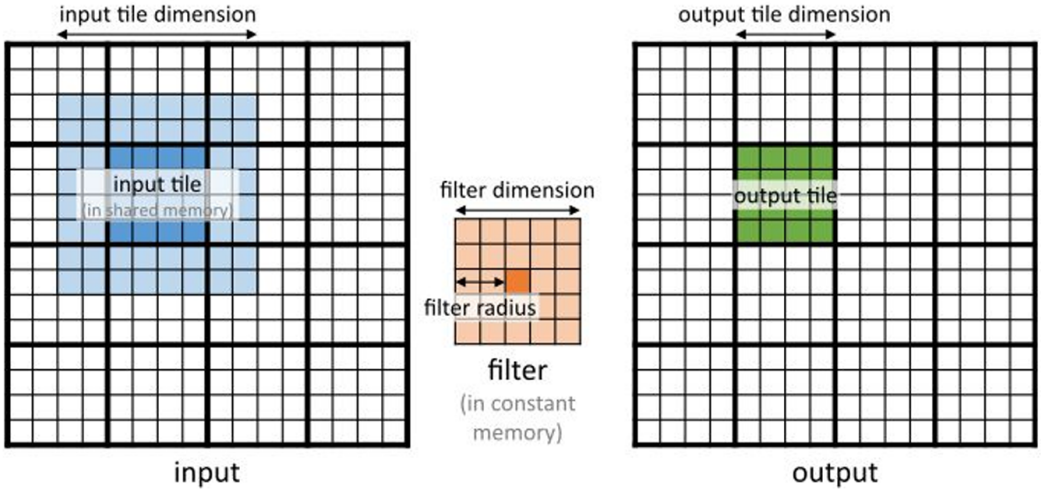
Since floating point operations are executed, 4 bytes are considered Memory Access:

$$\text{IN\_TILE\_DIM}^2 \cdot 4 = (\text{OUT\_TILE\_DIM} + 2 \cdot \text{FILTER\_RADIUS})^2 \cdot 4$$

Asymptotically, if $\text{OUT\_TILE\_DIM} \gg \text{FILTER\_RADIUS}$:

$$\frac{\text{Operations}}{\text{Memory Access}} = \frac{(2 \cdot \text{FILTER\_RADIUS} + 1)^2}{2}$$

Using a shared memory approach with tiling significantly alters how data is accessed and processed on a GPU, emphasizing efficiency and reduced global memory dependency. This method is particularly effective for operations like convolution where data reuse is high. The approach minimizes the latency of data access and maximizes the throughput by leveraging the fast shared memory on the GPU.



First Load the Tiles: Each thread in a block loads an element of the input image into shared memory, including necessary padding for the convolution operation. This reduces global memory accesses. After loading the data into shared memory, threads are synchronized to ensure all data is properly loaded before computation begins. cpp __shared__ input_type input_shared[IN_TILE_DIM][IN_TILE_DIM]; if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) { input_shared[tidy][tidx] = input[inRow * width + inCol]; } else { input_shared[tidy][tidx] = 0.0; } __syncthreads();

Compute Output Elements: Each thread computes an element of the output using the data in shared memory. This computation is localized to the data loaded by the block, significantly reducing the latency associated with memory access.

```
const int tileCol = tidx - FILTER_RADIUS
const int tileRow = tidy - FILTER_RADIUS

if (tileCol >= 0 && tileCol < OUT_TILE_DIM && tileRow >= 0 && tileRow < OUT_TILE_DIM) {
  input_type output_value{0.0f};
  #pragma unroll
  for (int row = 0; row < FILTER_SIZE; row++)
  #pragma unroll
    for (int col = 0; col < FILTER_SIZE; col++)
      output_value += constant_filter[row][col] * input_shared[tileRow + row][tileCol + col];
  output[inRow * width + inCol] = output_value;
}
```

Using shared memory can significantly reduce global memory accesses:

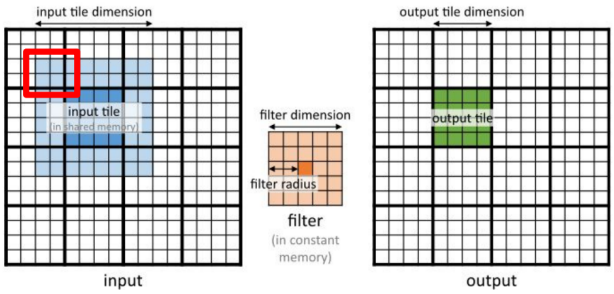
```
__global__ void convolution_tiled_kernel(const input_type *__restrict__ input,
                                         input_type *__restrict__ output,
                                         const int width, const int height) {
    __shared__ input_type input_shared[IN_TILE_DIM][IN_TILE_DIM];

    // Load data into shared memory
    const int inCol = blockIdx.x * OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
    const int inRow = blockIdx.y * OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
    if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
        input_shared[threadIdx.y][threadIdx.x] = input[inRow * width + inCol];
    } else {
        input_shared[threadIdx.y][threadIdx.x] = 0.0;
    }
    __syncthreads();

    // Compute output elements
    const int tileCol = threadIdx.x - FILTER_RADIUS;
    const int tileRow = threadIdx.y - FILTER_RADIUS;
    if (tileCol >= 0 && tileCol < OUT_TILE_DIM && tileRow >= 0 && tileRow < OUT_TILE_DIM) {
        input_type output_value = 0.0f;
        #pragma unroll
        for (int row = 0; row < FILTER_SIZE; row++) {
            #pragma unroll
            for (int col = 0; col < FILTER_SIZE; col++) {
                output_value += constant_filter[row][col] *
                                input_shared[tileRow + row][tileCol + col];
            }
        }
```

```
        output[inRow * width + inCol] = output_value;
    }
}
```

10.0.4 Caching Halo Cells

Halo cells are data that come from other tiles: upon running it is very likely that these data are already available in the L2 cache.

'



This technique loads only the internal part of the tile into shared memory, assuming halo cells are in L2 cache:

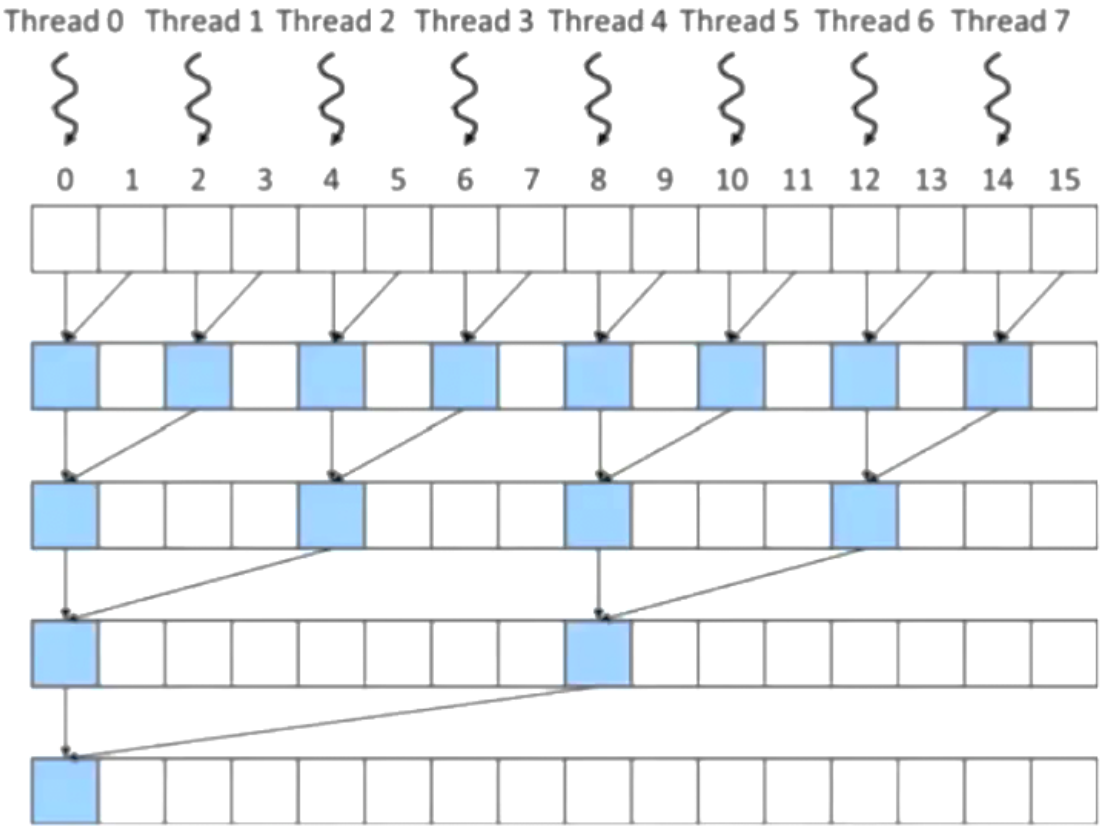
```
if (sharedRow >= 0 && sharedRow < TILE_DIM && sharedCol >= 0 && sharedCol < TILE_DIM) {
    // Shared memory access
    PValue += constant_filter[fRow][fCol] * input_shared[sharedRow][sharedCol];
} else {
    // Global memory access for halo cells
    int globalRow = row - FILTER_RADIUS + fRow;
    int globalCol = col - FILTER_RADIUS + fCol;
    if (globalRow >= 0 && globalRow < height && globalCol >= 0 && globalCol < width) {
        PValue += constant_filter[fRow][fCol] * input[globalRow * width + globalCol];
    }
}
```

Due to conditional statements handling halo and boundary conditions there is a **a lot of Branch Divergence**, which increase instruction count and reduce performance. **Compiler's Role:** the compiler generates a substantial number of instructions to manage this complexity, which can aggravate thread divergence.

11 Reduction

The concept of reduction involves collecting values from an array or collection and reducing them to a single value. Examples are the sum or finding the maximum value in an array.

The naive GPU implementation that divides the input into blocks of threads and has each thread process two elements. The threads then combine their results until only one thread remains, which performs the final reduction operation.



In this case the stride factor is two Essentially, it defines how far apart the memory locations each thread should read from or write to are spaced in each step of the reduction. In each subsequent step of the reduction process, the stride factor typically doubles, ensuring that each thread combines data from increasingly distant parts of the input array. This helps to efficiently reduce large arrays by combining results across the threads in a block, and eventually across multiple blocks, to arrive at a final result.

The value "two" you mentioned is likely the initial value for the stride factor, which indicates that initially, each thread reads data separated by one intervening element. The stride then doubles with each iteration of the reduction loop, allowing threads to progressively cover and reduce larger portions of data.

```
// CPU version of the reduction kernel
float reduce_cpu(const float* data, const int length) {
  float sum = 0;
  for (int i = 0; i < length; i++) { sum += data[i]; }
  return sum;
}
```

Each thread within a block performs a computation, accumulates values, and exchanges results with neighboring threads before storing it in global memory. The speaker highlights the importance of shifting input data between blocks using the stride factor and block index, ensuring each block processes different parts of the input.

The output will be a vector of double with the same dimension as the grid dimension, which is calculated by dividing the input dimension by the number of elements processed by each block. In the end, only the first thread of each block will have the final result after performing the reduction from all elements.

```
// GPU version of the reduction kernel
__global__ void reduce_gpu(double* __restrict__ input, double* __restrict__ output) {
  const unsigned int i = STRIDE_FACTOR * threadIdx.x;

  // Apply the offset
  input += blockDim.x * blockIdx.x * STRIDE_FACTOR;
  output += blockIdx.x;

  for (unsigned int stride = 1; stride <= blockDim.x; stride *= STRIDE_FACTOR) {
    if (threadIdx.x % stride == 0) {
      input[i] += input[i + stride];
    }
    __syncthreads();
  }

  // Write result for this block to global memory
  if (threadIdx.x == 0) {
    // You could have used only a single memory location and performed an atomicAdd
    *output = input[0];
  }
}
```

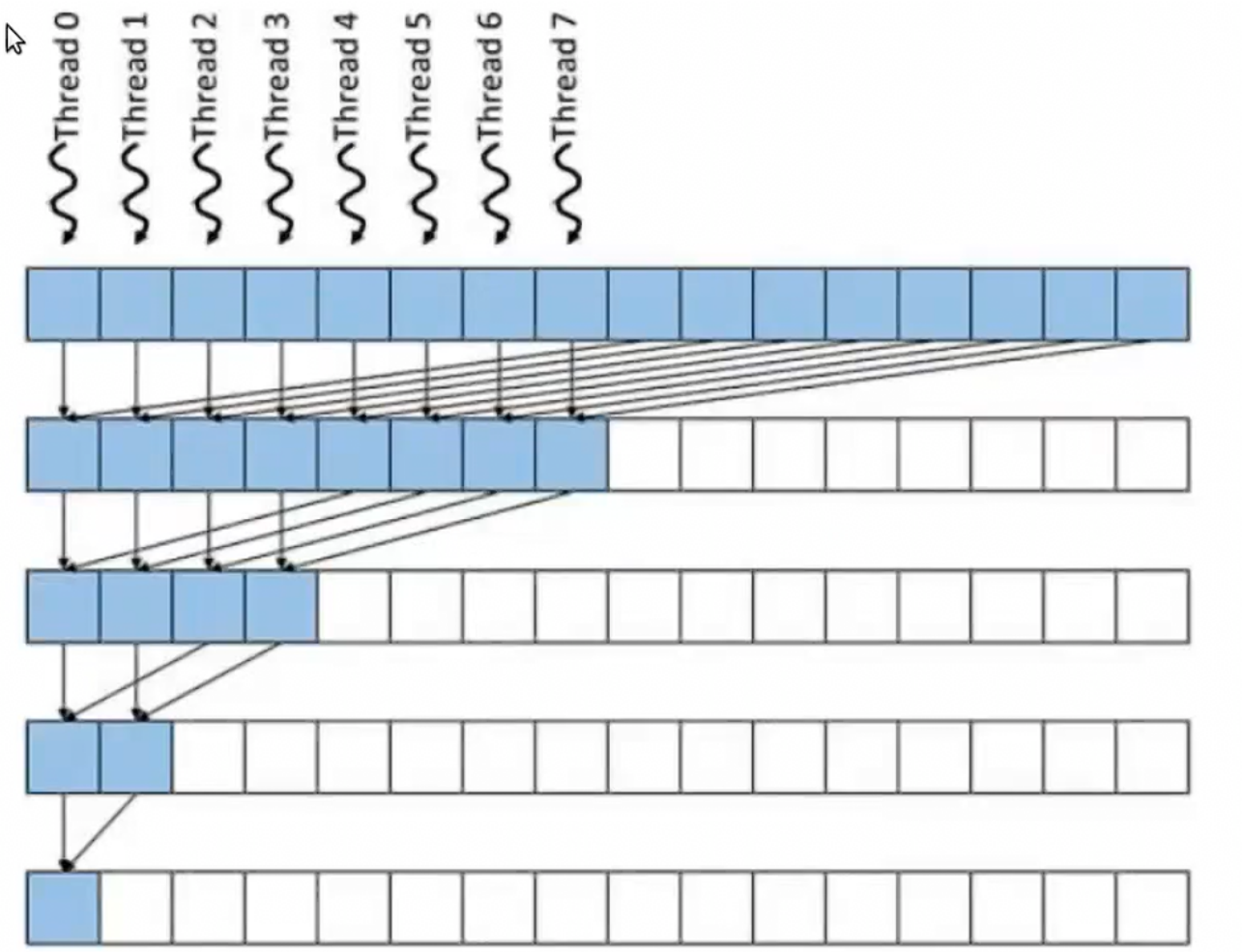
- `if (threadIdx.x % stride == 0)`: This condition checks if the current thread should participate in this iteration of the reduction. A thread participates if its index is a multiple of the current `stride`, meaning it is responsible for adding a specific element located `stride` positions away.

This current code relies heavily on global memory access with minimal use of cache levels.

Exploring ways to reduce memory transfer and improve memory locality is necessary.

11.0.1 Memory divergency

The key change is using the stride index in reverse order, dividing it by 2 at each iteration to maintain the same approach as before. An if condition is used to select which strides are active and which are not, based on the triad index being smaller than the actual value of the strides. This ensures that only necessary memory locations are accessed.




```
// Coalesced
__global__ void reduce_gpu(double* __restrict__ input, double* __restrict__ output) {
  const unsigned int i = threadIdx.x;

  // Apply the offset
  input += blockDim.x * blockIdx.x*STRIDE_FACTOR;
  output += blockIdx.x;

  for (unsigned int stride = blockDim.x; stride >= 1; stride /= STRIDE_FACTOR) {
    if (i < stride) {
      input[i] += input[i + stride];
    }
    __syncthreads();
  }

  // Write result for this block to global memory
  if (threadIdx.x == 0) {
    // You could have used only a single memory location and performed an atomicAdd
    *output = input[0];
  }
```

```
}
```

Coalesced Access -> Each thread in a warp accesses memory locations that are consecutive -> if thread 0 accesses memory address X, thread 1 accesses address X+1, thread 2 accesses X+2, and so on -> When threads in a warp access consecutive memory addresses, the GPU can combine these accesses into a single memory transaction -> 1. **Reduced Memory Transactions -> Improved Bandwidth Utilization -> Lower Latency**

11.0.2 Privatization

Privatization as another optimization technique, where instead of going back and forth between global memory, computations are performed in shared memory.

Threads write their partial sum result values to global memory o These values are reread by the same threads and others in the next iterations Shared memory has much shorter latency and higher bandwidth o It can fit the data required by the block's thread in this case

```
// Privatization
__global__ void reduce_gpu(const double* __restrict__ input, double* __restrict__ output) {
  __shared__ double partial[BLOCK_DIM];
  const unsigned int i = threadIdx.x;

  // Apply the offset
  input += blockDim.x * blockIdx.x*STRIDE_FACTOR;
  output += blockIdx.x;

  partial[i] = input[i] + input[i+BLOCK_DIM] //first iteration outside and LOAD

  for (unsigned int stride = blockDim.x / STRIDE_FACTOR; stride >= 1; stride /= STRIDE_FACTOR) {
    if (i < stride) {
      partial[i] += partial[i + stride];
    }
    __syncthreads();
  }

  // Write result for this block to global memory
  if (threadIdx.x == 0) {
    // You could have used only a single memory location and performed an atomicAdd
    *output = partial[0];
  }
}
```

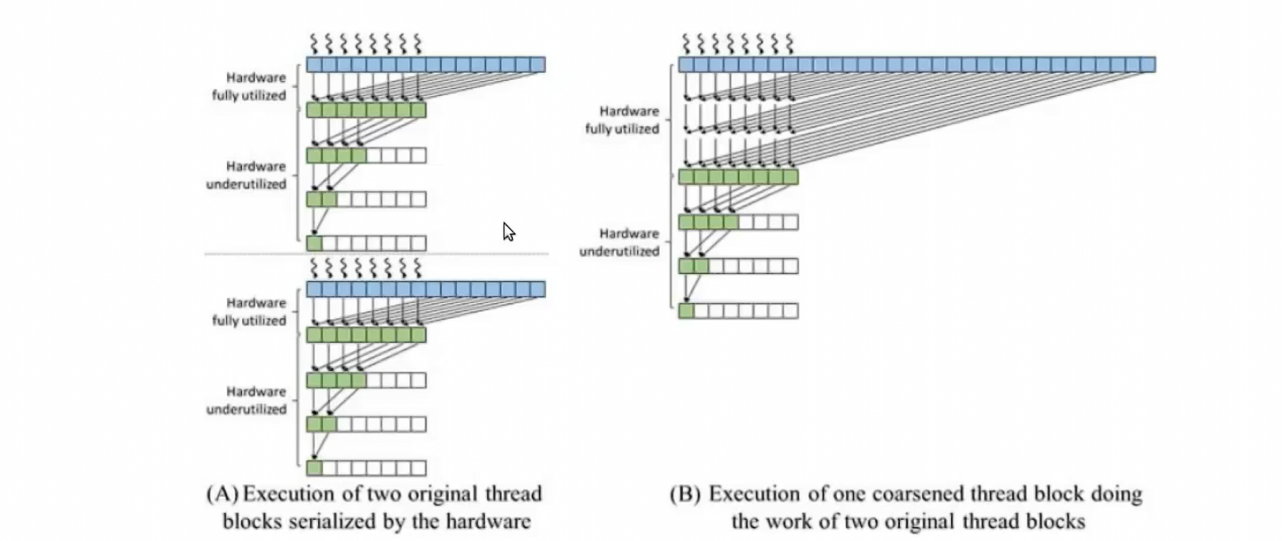
The use of shared memory allows for faster access times compared to global memory.

the first step of the reduction before entering the loop. This step effectively halves the number of active threads needed in the subsequent steps by pre-summing two distant elements of the input array. This is a common technique in reduction kernels where the first step is explicitly handled to reduce the complexity of the loop and decrease the number of iterations required.

To maximize memory bandwith and reducing workload (fewer iterations and threads) the initial step where data is copied from global to shared memory is not just about moving data around but an opportune moment to perform an initial partial sum of the data.

11.0.3 Thread coarsening

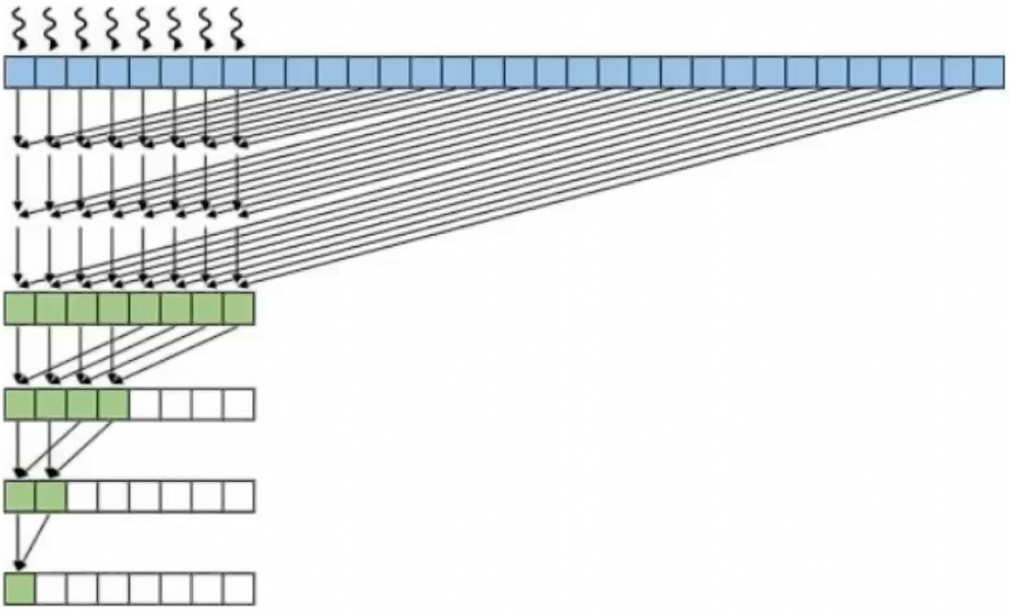
To reduce overhead, thread coarsening can be employed, allowing for more efficient use of hardware resources. Actually two solutions depending on the `COARSE_FACTOR`:



(A) Execution of two original thread blocks serialized by the hardware

(B) Execution of one coarsened thread block doing the work of two original thread blocks

Regarding spawning fewer threads:



The method involves serializing these thread blocks ourselves in a more efficient manner than the hardware would. The segment for each block is identified by multiplying with the COARSE FACTOR, and threads are tasked with more than just adding two elements.

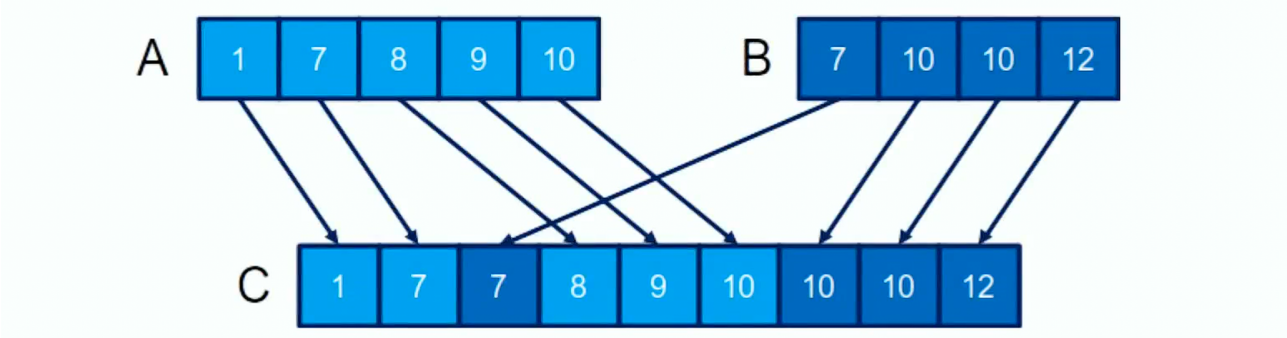
- Threads begin by calculating their own unique starting point in the input array by adjusting for block and thread indices multiplied by the `COARSE_FACTOR`.
- Each thread sums multiple elements from the input array, determined by the `COARSE_FACTOR`

- This sum is initially calculated using regular registers due to their faster access times compared to shared memory, although this approach increases register pressure which can potentially reduce parallelism.
- Subsequent reduction of these sums is performed in shared memory, consolidating the data further through synchronized strides, reducing the number of active threads in each step, and enhancing memory access patterns within the GPU's architecture.

```
// GPU version of the reduction kernel
__global__ void reduce_coarsening_gpu(const double* __restrict__ input, double* __restrict__ output) {
  __shared__ double input_s[BLOCK_DIM];
  const unsigned int t = threadIdx.x;

  // Apply the offset
  // NOTE: input const means that the content is const, the pointer can change
  input += blockDim.x * blockIdx.x * COARSE_FACTOR;
  output += blockIdx.x;

  double sum = input[t];
  // Here the hardware is fully utilized
  for (unsigned int tile = 1; tile < COARSE_FACTOR; ++tile) sum += input[t + tile * BLOCK_DIM];
  // You could have used directly the shared memory
  // Registers are faster though
  // High register pressure leads to lower parallelism
  input_s[t] = sum;

  // Perform reduction in shared memory
  for (unsigned int stride = blockDim.x / STRIDE_FACTOR; stride >= 1; stride /= STRIDE_FACTOR) {
    __syncthreads();
    if (t < stride) {
      input_s[t] += input_s[t + stride];
    }
  }

  // Write result for this block to global memory
  if (threadIdx.x == 0) {
    // You could have used only a single memory location and performed an atomicAdd
    *output = input_s[0];
  }
}
```

Key takeaways include the significant impact of adjusting the coarse factor and block size on performance, the benefits of using shared memory for accessing input elements which enhances parallel processing efficiency, and the optimization of reduction operations in global memory through atomic operations or synchronization mechanisms. The balance between high register usage and memory efficiency is crucial for optimizing GPU kernel performance.

12 Merge

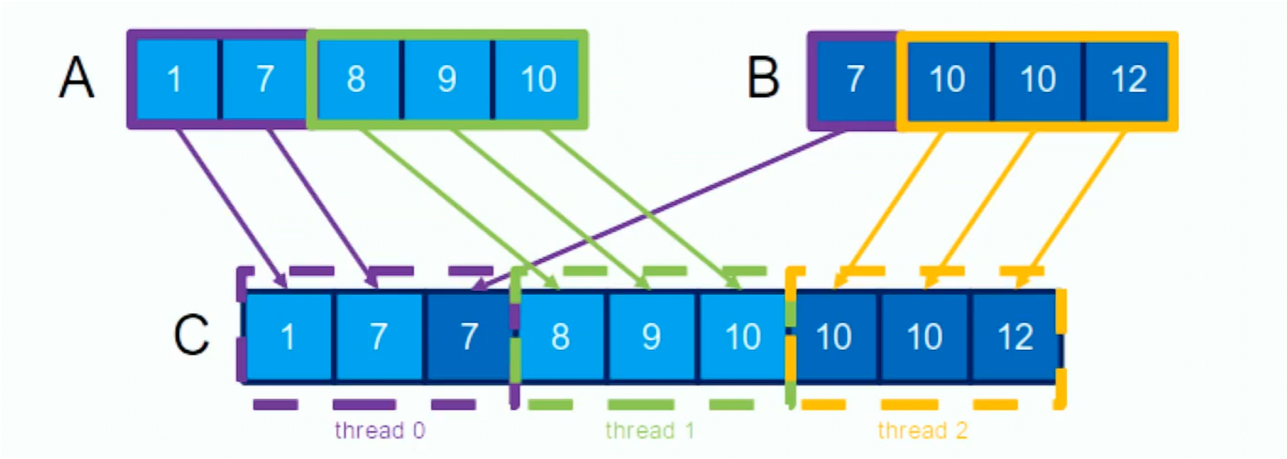



```c
// Function for CPU merge computation
void merge_cpu(const int *A, const int dim_A, const int *B, const int dim_B, int *C) {
  int i = 0;
  int j = 0;
  int k = 0;

  while ((i < dim_A) && (j < dim_B)) {
    if (A[i] <= B[j])
      C[k++] = A[i++];
    else
      C[k++] = B[j++];
  }
  if (i == dim_A) {
    while (j < dim_B) { C[k++] = B[j++]; }
  } else {
    while (i < dim_A) { C[k++] = A[i++]; }
  }
}
```

Merge Sort can be optimized for parallel execution, but it's not just a matter of re-implementing a classical algorithm; it involves strategic data management and computation distribution. Merge sort adheres to a stable sort criterion where identical elements from A precede those from B in list C, preserving the input order.

Naive

The basic approach divides the output list C among multiple threads, with each thread responsible for merging a specific section from lists A and B. This division is dynamic, depending on the specific elements of C that each thread is calculating, which can lead to uneven workload distribution among the threads.



Co rank

Array A (length 15):

56	279	359	365	377	466	482	598	655	671	704	726	767
954	973											

Array B (length 25):

16	25	99	115	175	178	185	197	308	390	411	439	450
468	540	575	620	640	640	838	853	945	952	964	971	

A more sophisticated method involves calculating the "co-rank" of elements, which helps in determining how input elements from A and B are paired to form the output C. The co-rank, defined for an element $C[k]$, is derived from indices i and j such that $i + j = k$.

$$A[i-1] \leq B[j]$$
$$B[j-1] < A[i]$$

The co-rank of an element in the output array (C) from a merge operation helps to determine precisely which elements from the two input arrays (A) and (B) will combine to form that specific output element. Specifically, if you are considering an element (C[k]) in the output array, the co-ranks (i) and (j) from arrays (A) and (B) respectively, satisfy the condition (k = i + j). This means that to compute the value of (C[k]), you need to merge elements up to (i) from array (A) and up to (j) from array (B).

By utilizing binary search based on co-rank, each thread can quickly determine the exact position (indices (i) and (j)) in the input arrays (A) and (B) that it needs to process.

This method uses a binary search to find the appropriate indices efficiently, optimizing the merge process with a complexity of $O(logN)$.

```
__device__ int co_rank(const int k, const int *__restrict__ A, const int m, const int *__restrict__ B, co
    int i = min(k, m);   // Start i at the smaller of k or m
```

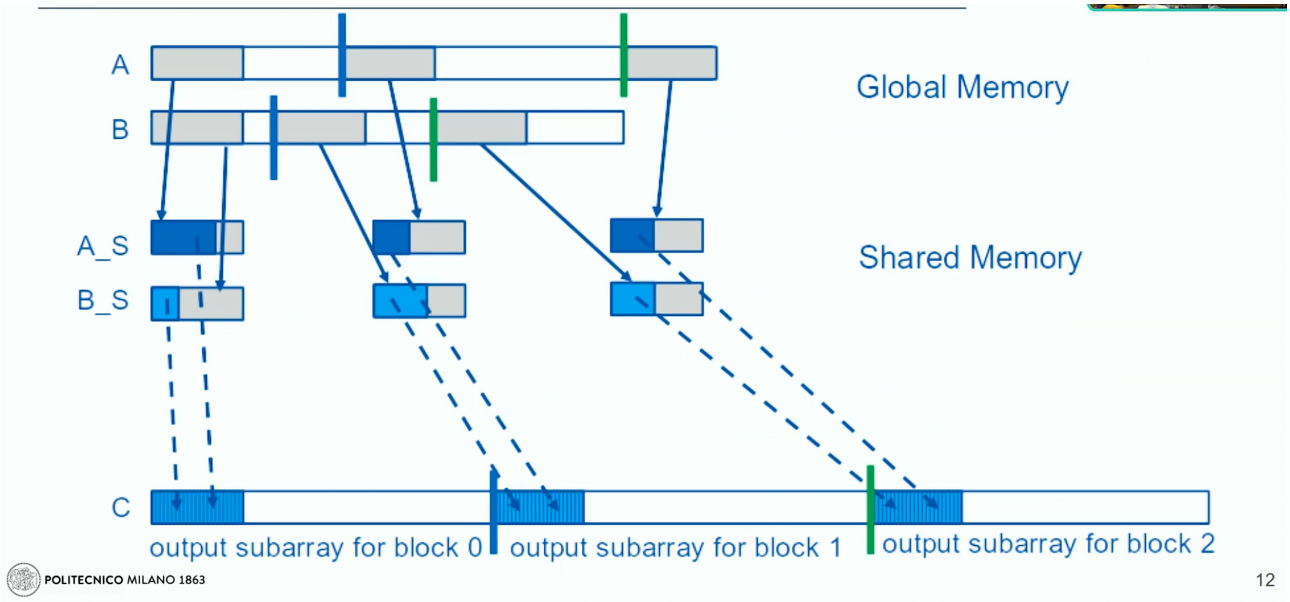
```
    int j = k - i;        // Calculate j based on i
    int i_low = max(0, k - n);  // Lower bound for i
    int j_low = max(0, k - m);  // Lower bound for j
    int delta;
    bool active = true;

    while (active) {
        if (i > 0 && j < n && A[i - 1] > B[j]) {
            // If A[i-1] is greater than B[j], adjust i and j
            // Ensure delta is at least 1 and calculate half the interval
            delta = max(1, (i - i_low) / 2);
            i -= delta;
            j += delta;
        } else if (j > 0 && i < m && B[j - 1] >= A[i]) {
            // If B[j-1] is greater than or equal to A[i], adjust i and j
            // Ensure delta is at least 1 and calculate half the interval
            delta = max(1, (j - j_low) / 2);
            i += delta;
            j -= delta;
        } else {
            // If neither condition is met, we've found the correct i
            active = false;
        }
    }

    return i;  // Return the co-rank i
}
```

With clear co-rank values, each thread or thread block can independently compute a segment of the output array without needing to wait for other threads to complete their tasks.

In practical scenarios, the input arrays A and B might not be of equal size, which can complicate the parallel processing of data. Co-rank helps manage these irregularities by providing a systematic way to allocate elements of A and B to threads regardless of their lengths.

Further optimizations include the use of shared memory and registers.

12.0.0.1 Tiling variant Each iteration of the merge operation involves loading specific segments ("tiles") of lists A and B into shared memory. In this strategy, during the first iteration, segments from both lists are uploaded into shared memory. All threads then evaluate which parts of these segments they need to merge based on their specific task within the overall merge operation.



In subsequent iterations, new segments of A and B are loaded into shared memory. It's crucial to note that the entire shared memory may not always be filled if the segments from A and B do not align perfectly, potentially leading to inefficiencies and the introduction of bugs.

12.0.0.2 Circular buffering One proposed enhancement involves using a circular buffering method to optimize the utilization of shared memory across iterations. Instead of reloading data that has already been brought into shared memory, the algorithm can maintain a dynamic index that tracks where new data should be written and read within the shared memory. This approach reduces the redundancy of memory accesses and maximizes the use of already-loaded data.

- If `A` and `B` are the input arrays, and `k` is the index in the merged array `C`:
  - `co_rank(k, A, dim_A, B, dim_B)` finds the position in `A` such that the sum of elements from `A` and `B` up to this position is equal to `k`.

Let's visualize the co-rank for `k = 2` in our example:

- `A = [1, 3, 5, 7]`
- `B = [2, 4, 6, 8]`

We want to find how many elements from `A` and `B` make up the first 2 elements in `C`:

- The co-rank splits the merged array such that the first `k` elements are from `A` and `B`.

In this case:

- `co_rank(2, A, 4, B, 4)` would find that:
  - 1 element from `A` (1) and 1 element from `B` (2) make up the first 2 elements in `C`.

After loading the data into the shared memory buffers, we need to update the starting index for the next iteration of the circular buffer:

```
Copy code
A_S_start = (A_S_start + A_S_consumed) % TILE_SIZE;
B_S_start = (B_S_start + B_S_consumed) % TILE_SIZE;
```

- `A_S_start`: The starting index in the circular buffer A_s.
- `A_S_consumed`: The number of elements consumed (loaded) in the current iteration for `A`.

13 Reduction and scan

13.1 Reduction

The reduction example demonstrates thread coarsening:

```
__global__ void reduce_coarsening_gpu(const double* __restrict__ input, double* __restrict__ output) {
  __shared__ double input_s[BLOCK_DIM];
  const unsigned int t = threadIdx.x;

  input += blockDim.x * blockIdx.x * COARSE_FACTOR;
  output += blockIdx.x;

  double sum = input[t];
  for (unsigned int tile = 1; tile < COARSE_FACTOR; ++tile)
    sum += input[t + tile * BLOCK_DIM];
  input_s[t] = sum;

  // ... rest of reduction
}
```

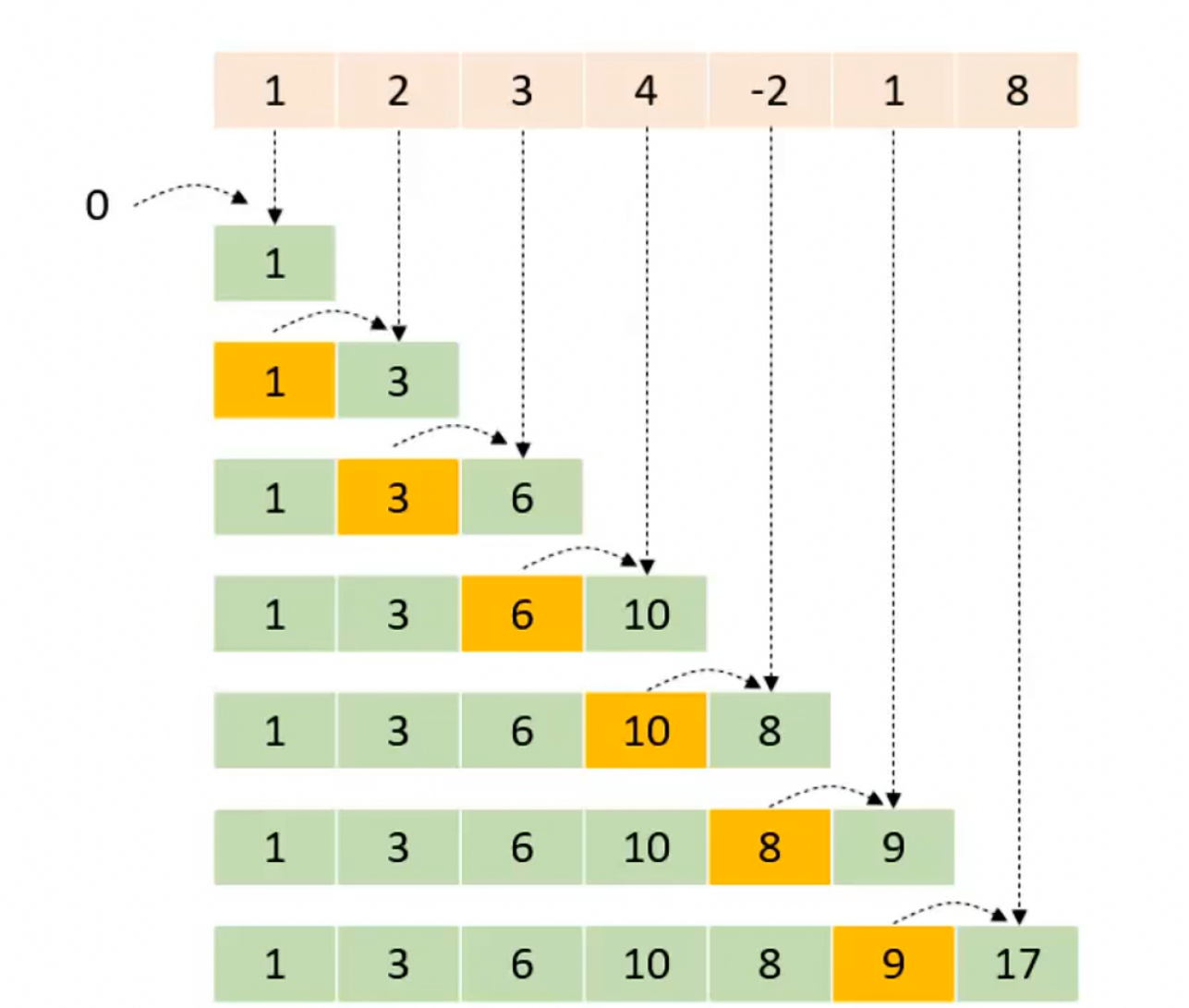
Here, each thread processes COARSE_FACTOR elements instead of just one, reducing the total number of threads and improving efficiency.

```
// GPU version of the reduction kernel
__global__ void reduce_coarsening_gpu(const double* __restrict__ input, double* __restrict__ output) {
  __shared__ double input_s[BLOCK_DIM];
  const unsigned int t = threadIdx.x;

  // Apply the offset
  // NOTE: input const means that the content is const, the pointer can change
  input += blockDim.x * blockIdx.x * COARSE_FACTOR;
  output += blockIdx.x;

  double sum = input[t];
  // Here the hardware is fully utilized
  for (unsigned int tile = 1; tile < COARSE_FACTOR; ++tile) sum += input[t + tile * BLOCK_DIM];
  // You could have used directly the shared memory
  // Registers are faster though
  // High register pressure leads to lower parallelism
  input_s[t] = sum;

  // Perform reduction in shared memory
  for (unsigned int stride = blockDim.x / STRIDE_FACTOR; stride >= 1; stride /= STRIDE_FACTOR) {
    __syncthreads();
    if (t < stride) {
      input_s[t] += input_s[t + stride];
    }
  }

  // Write result for this block to global memory
  if (threadIdx.x == 0) {
    // You could have used only a single memory location and performed an atomicAdd
    *output = input_s[0];
```

```
  }
}
```

13.2 Scan

A scan operation, also known as a prefix sum, is similar to reduction but produces an array of results instead of a single value.



**Two types of scan:

- **Inclusive scan**: includes current element in partial reduction.
- **Exclusive scan**: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element.

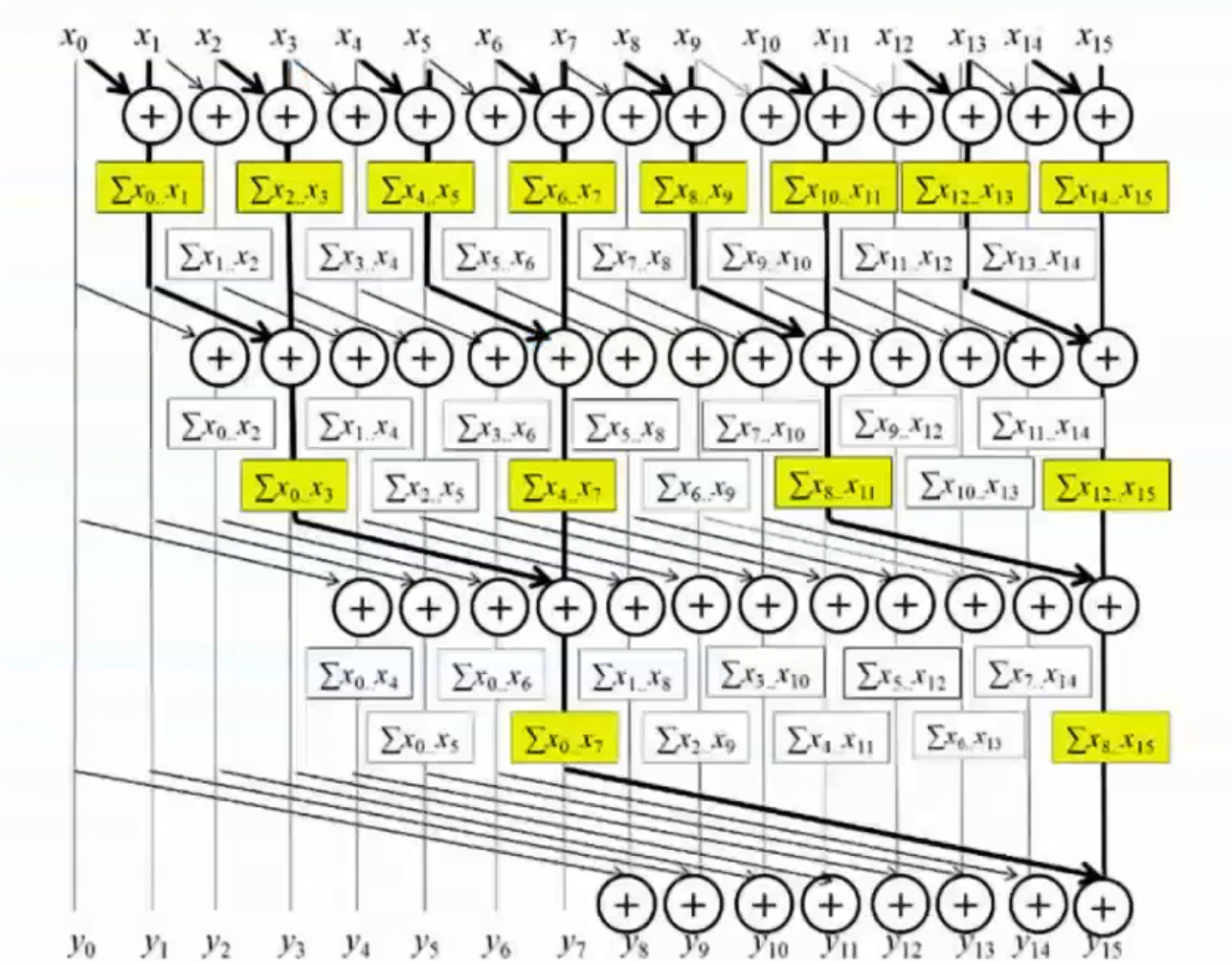
The choice between inclusive and exclusive scans can affect how the parallelization strategy is implemented due to the initial value settings and propagation of sum values through the dataset.

```cpp
// CPU version of the scan kernel EXCLUSIVE
void scan_cpu(const float* input, float* output, const int length) {
  output[0] = 0;
  for (int i = 1; i < length; ++i) { output[i] = output[i - 1] + input[i - 1]; }
}

// Difference with reduction operation:
float reduce_cpu(const float* data, const int length) {
  float sum = 0;
  for (int i = 0; i < length; i++) { sum += data[i]; }
  return sum;
}
```

There is the risk to try to parallelize a sequential version which has the bottleneck of the longest path (leading to $O(n^2)$): each thread has to look at all previous elements, leading to redundant work and poor utilization of the GPU's algorithmic capabilities:

```cpp
__global__ void
    naive_scan_gpu(const float* __restrict__ input, float* __restrict__ output, const int length) {
  const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
  float sum{0};
  for (unsigned int i = 0; i < index; i++) { sum += input[i]; }
  output[index] = sum;
  return;
}
```

The Kogge-Stone algorithm is used for efficiently summing elements in parallel on a GPU. Each thread sums a range of elements and then combines these sums with results from other threads. *Basically I increase the parallelization with a tradeoff of synchronization.*




```
// GPU version of the scan kernel EXCLUSIVE Kogge Stone
__global__ void
    kogge_stone_scan_gpu(const float* __restrict__ input, float* __restrict__ output, const int length) {
  __shared__ float input_s[BLOCK_DIM];
  const unsigned int tid = threadIdx.x;

  input_s[tid] = input[tid - 1];

  for (unsigned int stride = 1; stride < length; stride *= STRIDE_FACTOR) {
    __syncthreads();
    float temp;
    if (tid >= stride)
      temp = input_s[tid] + input_s[tid - stride];
    __syncthreads();
    if (tid >= stride)
      input_s[tid] = temp;
  }
  output[tid] = input_s[tid];

  return;
```

}

- **Initial State**: Each `output[i]` starts with the corresponding input element x.
- **Iterations**: After `k` iterations, `output[i]` will contain the sum of `2^k` input elements.
- **Thread ID Adjustment**: `input_s[tid] = input[tid - 1];` initializes each thread with the input shifted by one index back. This is critical for managing edge cases where `tid = 0`.
- **Loop for Summation**: The loop increases the `stride` geometrically, reducing the number of active threads at each step. This reduction is typical in parallel reduction algorithms to minimize active threads as the problem size effectively decreases.
- **Synchronization**: Twice per loop iteration to avoid hazards:
  - Before calculating the temporary sum to ensure all threads have the correct values (`__syncthreads()`).
  - After storing the temporary sum to ensure that no writes occur until all calculations in the current stride are complete (`__syncthreads()`).

note that in this version - All threads must wait at synchronization points to ensure data integrity, which can introduce delays or "stalls" where threads are idle, waiting for others to reach the synchronization point.

Another version is the improved Kogge-Stone algorithm using a double buffering technique: double the shared memory for each block to alternate between reading input and writing output across iterations. This technique, known as "ping-pong", alternates the roles of input and output buffers, reducing the need for synchronization barriers typically required in single-buffer implementations.

```
// Double buffering approach scan
__global__ void
    kogge_stone_scan_gpu(const float* __restrict__ input, float* __restrict__ output, const int length) {
  __shared__ float input_s[BLOCK_DIM * 2];
  const unsigned int tid = threadIdx.x;
  unsigned int pout = 0, pin = 1;

  input_s[tid]          = input[tid];
  input_s[length + tid] = input_s[tid];

  __syncthreads();
  for (unsigned int stride = 1; stride < length; stride *= STRIDE_FACTOR) {
    pout = 1 - pout; // swap double buffer indices
    pin  = 1 - pin;
    if (tid >= stride)
    //making reduction
    input_s[pout * length + tid] = input_s[pin * length + tid] + input_s[pin * length + tid - stride];
    else
    //otherwise simply copy the previous value
    input_s[pout * length + tid] = input_s[pin * length + tid];
    __syncthreads();
  }
  output[tid] = input_s[pout * length + tid];

  return;
}
```

- it's a way to overlap computation and memory transfers
- In one iteration, a thread reads from one buffer (input) and writes the result to the other buffer (output).
- In the next iteration, the roles of the buffers are switched: the previous output buffer becomes the new input buffer, and vice versa.
- This swapping allows continuous use of data without waiting for other threads to sync up, thereby

minimizing synchronization points.

The double buffering approach minimizes the need for `__syncthreads()` calls, which are typically used to prevent read-after-write hazards. This results in fewer stalls and more continuous computation.

14 Stencil

A stencil in computational terms refers to a pattern of computing values on a grid where each point's value is determined based on the values of its neighboring points. This is similar to the process of convolution, often used in image processing, where a matrix (kernel) is slid over data to compute outputs based on the kernel's weighted sum of the input values it covers.

```
void stencil_cpu(const float *in, float *out, const int N) {
  for (int i = 1; i < N - 1; ++i)
    for (int j = 1; j < N - 1; ++j)
      for (int k = 1; k < N - 1; ++k)
        get(out, i, j, k, N) =
                                c0 * get(in, i, j, k, N)
                                + c1 * get(in, i, j, k - 1, N) +
                               c2 * get(in, i, j, k + 1, N)
                               + c3 * get(in, i, j - 1, k, N)
                               + c4 * get(in, i, j + 1, k, N)
                               + c5 * get(in, i - 1, j, k, N) +
                               c6 * get(in, i + 1, j, k, N);
}
```

Stencil computations are crucial in fields like computational fluid dynamics, heat conductance, weather forecasting, and electromagnetics. Stencils are often used to approximate derivative values from discrete data points that represents physical quantities.

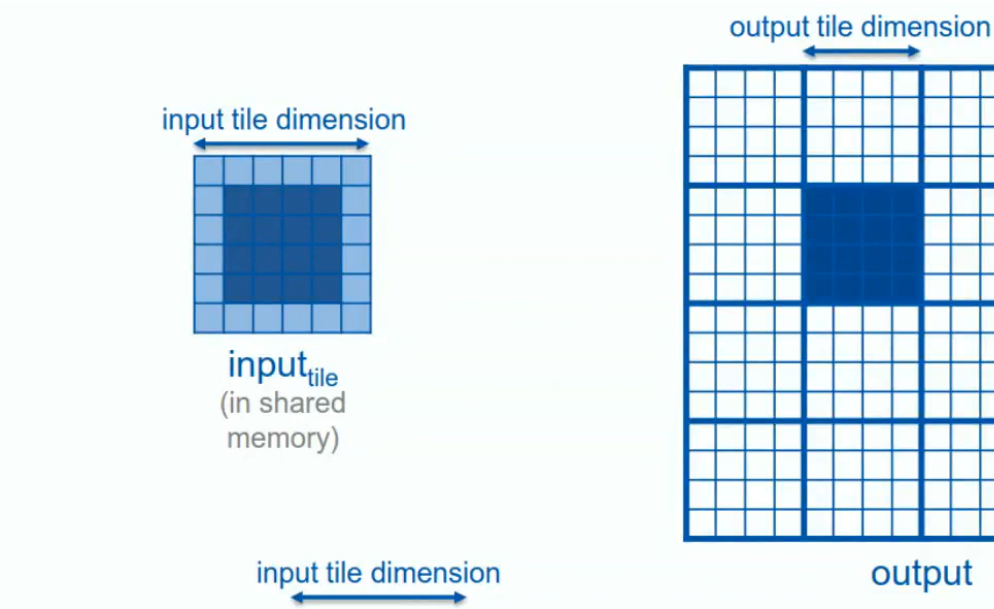
- **Naive Implementation**: Assign one GPU thread per grid point. Each thread calculates the output for a point without needing to wait for other threads, making the process highly parallel.

```
// Kernel function for GPU
__global__ void stencil_kernel_gpu(const float *__restrict__ in, float *__restrict__ out, const int N) {
  const unsigned int i = blockIdx.z * blockDim.z + threadIdx.z;
  const unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
  const unsigned int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
    get(out, i, j, k, N) =
                             c0 * get(in, i, j, k, N)
                             + c1 * get(in, i, j, k - 1, N)
                            c2 * get(in, i, j, k + 1, N)
                            + c3 * get(in, i, j - 1, k, N)
                            + c4 * get(in, i, j + 1, k, N)
                            + c5 * get(in, i - 1, j, k, N) +
                            c6 * get(in, i + 1, j, k, N);
  }
}
```

Several strategies can improve performance and efficiency:

- **Tiling and Privatization**: Use shared memory on the GPU to reduce the frequency of data transfer between the GPU and its main memory.

- **Coarsening and Slicing**: Adjust the organization of threads and data to better fit the GPU's architecture, enhancing performance without exceeding memory capacities.
- **Register Tiling**: Use GPU registers to store data temporarily, reducing the reliance on shared memory and thus saving space.



14.0.0.1 Tiling and Privatization

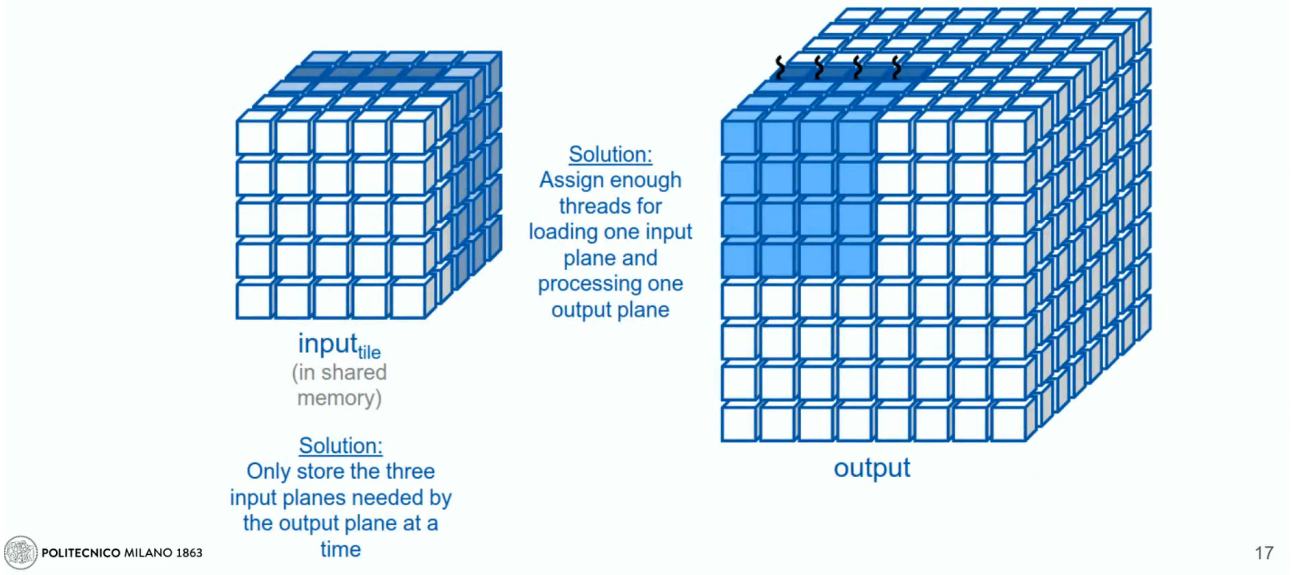
Differences:

- each thread still corresponds to a cell in the grid, much like in the naive version
- Before performing computations, this kernel loads necessary data into a shared memory array (`in_s`)

```
// Tiling and Privatization
__global__ void stencil_kernel_tiling_gpu(const float *__restrict__ in, float *__restrict__ out, const in
  const unsigned int i = blockIdx.z * OUT_TILE_DIM + threadIdx.z - 1;
  const unsigned int j = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
  const unsigned int k = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;

  __shared__ float in_s[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];
  // Greater than 0 not required since unsigned int are used
  if (i < N && j < N && k < N) {
    in_s[threadIdx.z][threadIdx.y][threadIdx.x] = get(in, i, j, k, N);
  }
  __syncthreads();

  if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
    if (threadIdx.z >= 1 && threadIdx.z < IN_TILE_DIM - 1 && threadIdx.y >= 1 &&
        threadIdx.y < IN_TILE_DIM - 1 && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
      get(out, i, j, k, N) = c0 * in_s[threadIdx.z][threadIdx.y][threadIdx.x] +
                             c1 * in_s[threadIdx.z][threadIdx.y][threadIdx.x - 1] +
                             c2 * in_s[threadIdx.z][threadIdx.y][threadIdx.x + 1] +
                             c3 * in_s[threadIdx.z][threadIdx.y - 1][threadIdx.x] +
                             c4 * in_s[threadIdx.z][threadIdx.y + 1][threadIdx.x] +
                             c5 * in_s[threadIdx.z - 1][threadIdx.y][threadIdx.x] +
                             c6 * in_s[threadIdx.z + 1][threadIdx.y][threadIdx.x];
```

```
    }
  }
}
```

14.0.0.2 Coarsening and Slicing This optimization strategy addresses the limitations posed by block sizes in GPU computations. By dividing the computation into slices and iterating over them, we enhance the use of the GPU's memory and processing capabilities.



1. **Thread Coarsening**: Instead of processing the entire 3D matrix at once, each thread block handles one slice of the x-y plane at a time. This iteration happens in the z-direction.
2. **Memory Management**: Only three slices of the input matrix are stored at any time: the previous, current, and next. This reduces memory requirements significantly.

- **Initial Setup**: 3 slices along the z-axis: the current, the previous and next slices into shared memory before starting the computation loop. After computation, update the shared memory to prepare for the next slice.
- **Efficiency**: This method minimizes the amount of shared memory needed and increases the arithmetic intensity, which is closer to the optimal value of 3.25 operations per byte (OP/B).
- **Memory Switching**: Similar to a buffer system, shared memory layers are rotated to continuously reuse and update data without frequent global memory access.

```
__global__ void
    stencil_kernel_coarsening_tiling_gpu(const float *__restrict__ in, float *__restrict__ out, const int
  const unsigned int i_start = blockIdx.z * Z_SLICING;
  const unsigned int j       = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
  const unsigned int k       = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;

  __shared__ float in_prev_s[IN_TILE_DIM][IN_TILE_DIM];
  __shared__ float in_curr_s[IN_TILE_DIM][IN_TILE_DIM];
  __shared__ float in_next_s[IN_TILE_DIM][IN_TILE_DIM];
  // Check greater than 0 not needed since index is an unsigned int
  if (i_start - 1 < N && j < N && k < N) {
    in_prev_s[threadIdx.y][threadIdx.x] = get(in, i_start - 1, j, k, N);
  }
```

```
  if (i_start < N && j < N && k < N) {
    in_curr_s[threadIdx.y][threadIdx.x] = get(in, i_start, j, k, N);
  }
  for (unsigned int i = i_start; i < i_start + Z_SLICING; ++i) {
    // Check greater than 0 not needed since index is an unsigned int
    if (i + 1 < N && j < N && k < N) {
      in_next_s[threadIdx.y][threadIdx.x] = get(in, i + 1, j, k, N);
    }
    __syncthreads();

    if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
      if (threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1 && threadIdx.x >= 1 &&
          threadIdx.x < IN_TILE_DIM - 1) {
        get(out, i, j, k, N) =
            c0 * in_curr_s[threadIdx.y][threadIdx.x]
            + c1 * in_curr_s[threadIdx.y][threadIdx.x - 1]
            + c2 * in_curr_s[threadIdx.y][threadIdx.x + 1]
            + c3 * in_curr_s[threadIdx.y - 1][threadIdx.x]
            + c4 * in_curr_s[threadIdx.y + 1][threadIdx.x]
            + c5 * in_prev_s[threadIdx.y][threadIdx.x]
            + c6 * in_next_s[threadIdx.y][threadIdx.x];
      }
    }
    __syncthreads();

    in_prev_s[threadIdx.y][threadIdx.x] = in_curr_s[threadIdx.y][threadIdx.x];
    in_curr_s[threadIdx.y][threadIdx.x] = in_next_s[threadIdx.y][threadIdx.x];
  }
}
```

14.0.0.3 Registers optimization

Reducing the use of shared and register memory: only store a single slice of the data grid at any iteration. By allocating the next slice's data elements to registers and only transferring them to shared memory when they become the current slice, then moving them back to registers once they become the previous slice.

```
__global__ void
    stencil_kernel_register_tiling_gpu(const float *__restrict__ in, float *__restrict__ out, const int N
  const unsigned int i_start = blockIdx.z * Z_SLICING;
  const unsigned int j       = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
  const unsigned int k       = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;

  float in_prev;
  __shared__ float in_curr_s[IN_TILE_DIM][IN_TILE_DIM];
  float in_next;
  // Check greater than not needed since index is an unsigned int
  if (i_start - 1 < N && j < N && k < N) {
    in_prev = get(in, i_start - 1, j, k, N);
  }
  if (i_start < N && j < N && k < N) {
    in_curr_s[threadIdx.y][threadIdx.x] = get(in, i_start, j, k, N);
  }
```

```
  for (unsigned int i = i_start; i < i_start + Z_SLICING; ++i) {
    // Check greater than not needed since index is an unsigned int
    if (i + 1 < N && j < N && k < N) {
      in_next = get(in, i + 1, j, k, N);
    }
    __syncthreads();
    if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
      if (threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1 && threadIdx.x >= 1 &&
          threadIdx.x < IN_TILE_DIM - 1) {
        get(out, i, j, k, N) =
            c0 * in_curr_s[threadIdx.y][threadIdx.x]
            + c1 * in_curr_s[threadIdx.y][threadIdx.x - 1]
            + c2 * in_curr_s[threadIdx.y][threadIdx.x + 1]
            + c3 * in_curr_s[threadIdx.y - 1][threadIdx.x]
            + c4 * in_curr_s[threadIdx.y + 1][threadIdx.x]
            + c5 * in_prev + c6 * in_next;
      }
    }
    __syncthreads();
    in_prev                              = in_curr_s[threadIdx.y][threadIdx.x];
    in_curr_s[threadIdx.y][threadIdx.x] = in_next;
  }
}
```

- **Single Slice in Shared Memory**: Only the current slice is maintained in shared memory, reducing the shared memory requirements significantly.
- **Register Usage**: Two additional registers per thread are used to store the previous and next slices' data, enabling quick access and modifications without repeatedly accessing shared memory.
- **Memory Switching**: The stencil computations involve rotating the data between registers and shared memory, reducing the frequency and cost of memory accesses.

Smaller shared memory usage -> scarse resources is not registers but shared memory

15 Graph Traversal

Before diving in graph traversal , let's talk on ways to efficiently store sparse matrix. Sparse matrix computation is crucial in scientific, engineering, and financial modelling problems, where the majority of matrix elements are zeros.

15.0.1 Sparse Matrix Formats

Storing and processing these zeros wastes memory and bandwidth. To address this, various sparse matrix formats have been developed to compactly represent the non-zero elements, influencing GPU performance.

These formats include:

- CSR (Compressed Sparse Row)
- COO (Coordinate Format)
- ELL (ELLpack Format)

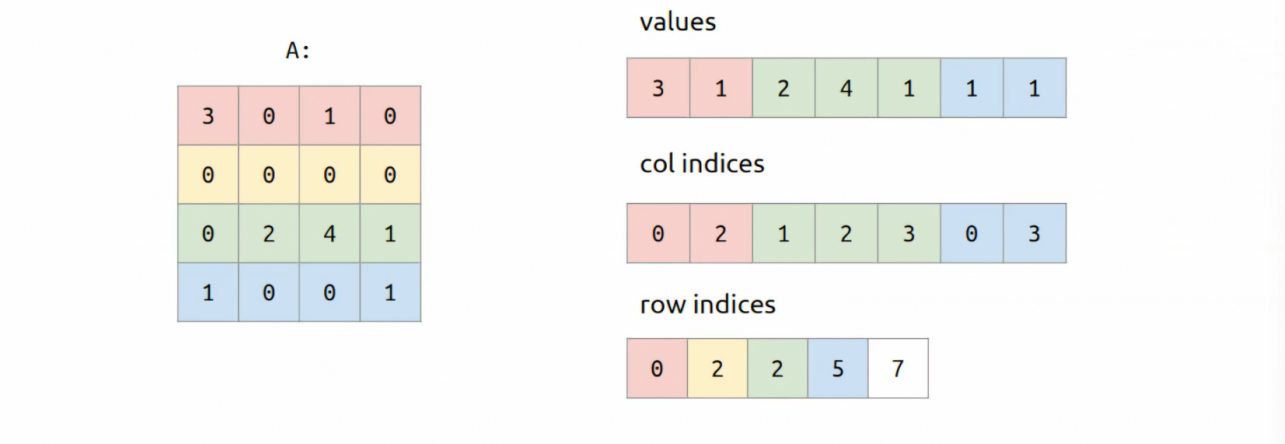
With this parameters:

- **M**: Number of rows in the matrix
- **N**: Number of columns in the matrix

- **K**: Number of nonzero entries in the densest row
- **S**: Sparsity level `[0-1]`, `1` being fully-dense

Format	Storage Requirements
Dense	MN
Compressed Sparse Row (CSR)	$2MNS + M + 1$
ELL	$2MK$
Coordinate (COO)	$3MNS$
Hybrid ELL / COO (HYB)	$2MK << 3MNS$

15.0.1.1 CSR The CSR format stores only the non-zero values a single vector: additional storage is required to locate the start of each row and the column of each element in the values vector.

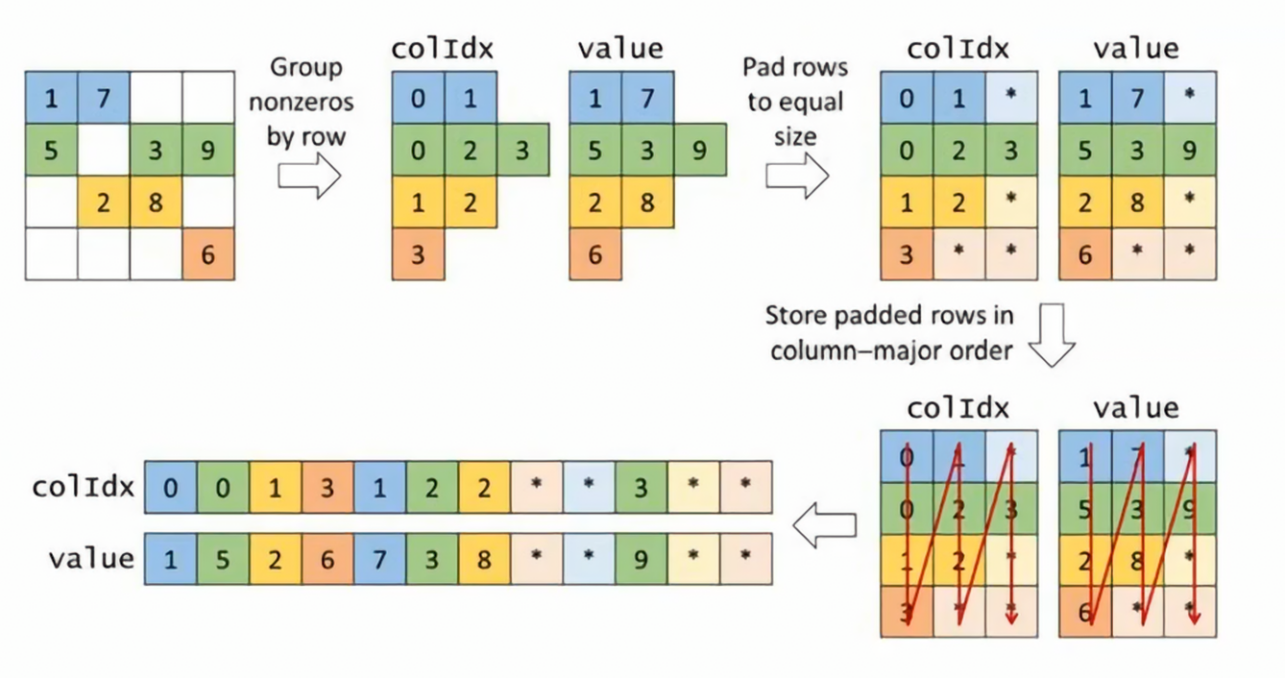


The space requirements are defined as $2MNS + M + 1$, and CSR is space-saving if $S < \left(1 - \frac{1}{N}\right)/2$.

Assigning a thread to each row for parallel SpMV (Sparse Matrix-Vector multiplication) ensures each thread handles a distinct output value, but consecutive threads accessing distant memory locations result in inefficient memory bandwidth utilization and potential control flow divergence.

15.0.2 ELL format

Originating from ELLPACK, relies on the maximum number of non-zero elements per row and **padding** elements. It's more flexible than CSR since non-zero elements can be added by replacing padding elements.



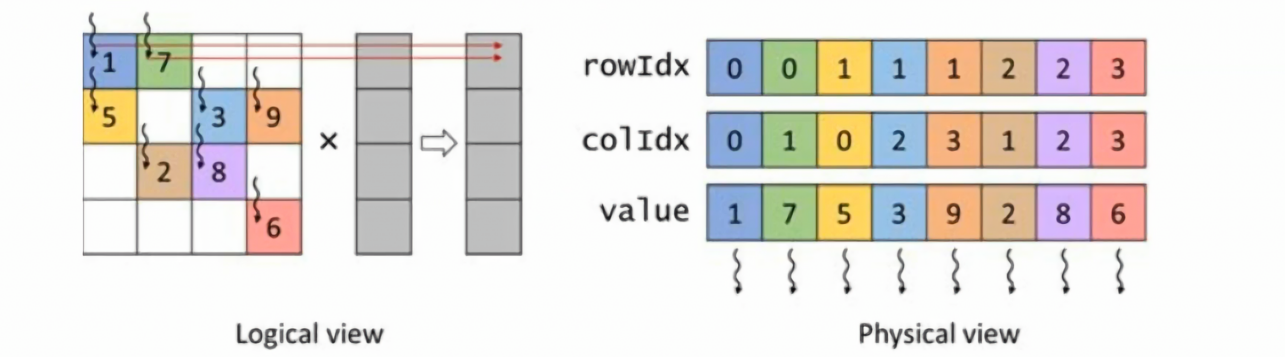
ELL uses memory bandwidth efficiently due to coalesced memory access but can lead to load imbalance. The space requirements for ELL are $2MK$, and it saves space if $K < \frac{N}{2}$.

15.0.3 COO

COO format stores both column and row indexes for every non-zero element, leading to higher space requirements but offering flexibility in adding non-zero elements.

Parallel SpMV in COO assigns each thread to compute a non-zero element, balancing the workload but requiring atomic operations for synchronization. The space requirements for COO are $3MNS$, and it saves space if $S < \frac{1}{3}$.

```
struct SparseMatrixCOO {
  int* row_indices;
  int* column_indices;
  float* values;
  int num_elements;
}
```




```
void SpMV_COO(const SparseMatrixCOO* A, const float* x, float* y) {
```

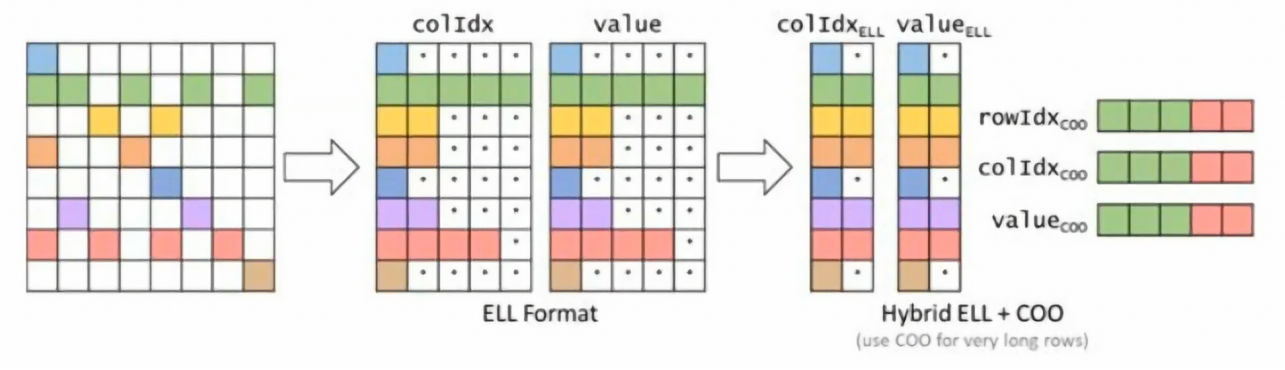
```
  for (int element = 0; element < A->num_elements; ++element) {
    const int column = A->column_indices[element];
    const int row    = A->row_indices[element];
    y[row] += A->values[element] * x[column];
  }
}
```

And in CUDA:

```
__global__ void SpMV_COO_gpu(
    const int* __restrict__ row_indices,
    const int* __restrict__ column_indices,
    const float* __restrict__ values,
    const int num_elements,
    const float* x,
    float* y) {

    for (int element = threadIdx.x + blockIdx.x * blockDim.x; element < num_elements;element += blockDim.
        const int column = column_indices[element];
        const int row    = row_indices[element];
        atomicAdd(&y[row], values[element] * x[column]);
      }
}
```

15.0.4 Hybrid ELL/COO

A hybrid ELL/COO format addresses ELL's inefficiency in handling rows with many non-zeros by placing these non-zeros in a COO format, leading to a more efficient ELL representation for the remaining elements.



This approach enhances space efficiency and flexibility, making it beneficial for iterative solvers despite the overhead of using two different formats. The code is basically the same but just combines the two previous methods.

15.0.5 Extra

Other formats include:

- **Jagged Diagonal Storage (JDS):**
  - Groups similarly dense rows into partitions.
  - Represents partitions independently using either CSR or ELL.
  - Sorts rows by density, avoiding padding and improving space efficiency.
  - Allows coalesced memory access but lacks flexibility in adding new elements.

- **Diagonal (DIA):**
  - Stores only a sparse set of dense diagonal vectors.
  - For each diagonal, the offset from the main diagonal is stored.
- **Packet (PKT):**
  - Reorders rows and columns to concentrate nonzeros into roughly diagonal submatrices.
  - Improves cache performance as nearby rows access nearby x elements.
- **Dictionary of Keys (DOK):**
  - Stores the matrix as a map from (row, column) index pairs to values.
  - Useful for building or querying a sparse matrix, but iteration is slow.
- **Compressed Sparse Column (CSC):**
  - Similar to CSR but stores a dense set of sparse column vectors.
  - Useful when column sparsity is much more regular than row sparsity.
- **Blocked CSR:**
  - Divides the matrix into blocks stored using CSR with the indices of the upper left corner.
  - Useful for block-sparse matrices.

In general, the FLOPS ratings achieved by both CPUs and GPUs are much lower for sparse matrix computation than for dense matrix computation. For example, in SpMV (Sparse Matrix-Vector multiplication) computation, there is no data reuse in the sparse matrix, which results in a low operational intensity. The operational intensity (OP/B) is essentially 0.25, significantly limiting the achievable FLOPS rate to a small fraction of the peak performance compared to dense matrix computations.

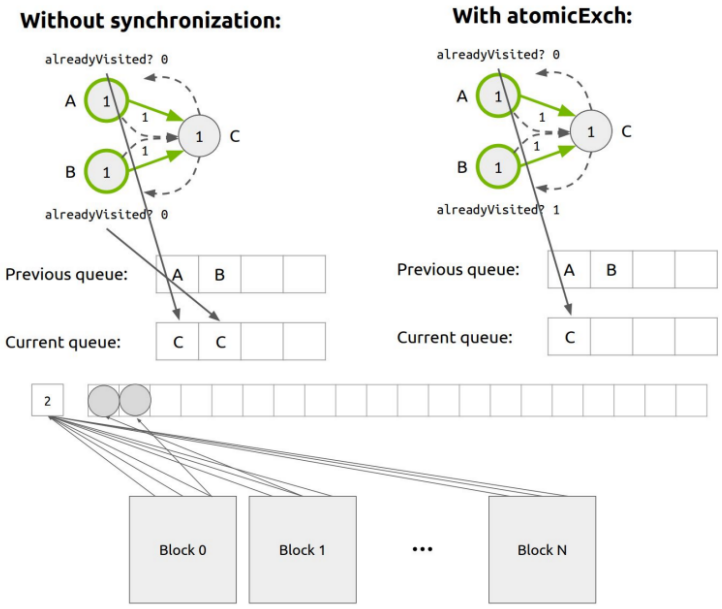
15.1 Graph traversal

Graphs are intrinsically related to sparse matrices, with an intuitive representation being an adjacency matrix. Consequently, graph computation can be formulated in terms of sparse matrix operations:

GPU Solutions for graph Traversal are:

- **Iterative Vertex Assignment**
  - Method: Iterate or assign each thread to a vertex.
    - For each iteration, check all incoming edges to see if the source vertex was visited in the last iteration.
    - If visited, mark the vertex as visited in the current iteration.
  - Efficiency: Not very work efficient.
    - Complexity: (O(VL)), where (V) is the number of vertices and (L) is the length of the longest path.
  - Challenge: Difficult to detect a stopping criterion.
- **Frontier-Based Thread Assignment**
  - Method: Assign threads to frontier vertices from the previous iteration.
    - Add all non-visited neighbors to the next frontier.
    - The source will be the first element in the frontier.
  - Parallel Execution: Threads execute in parallel. - Issue: A variable number of unnecessary threads are launched. - Explanation: This happens because the frontier size can vary greatly between iterations, leading to **imbalanced workloads** among threads. Efficient load balancing in such dynamic scenarios is challenging, resulting in the launch of more threads than necessary to ensure all possible work is covered.

Regarding frontier output interface:



POLITECNICO MILANO 1863

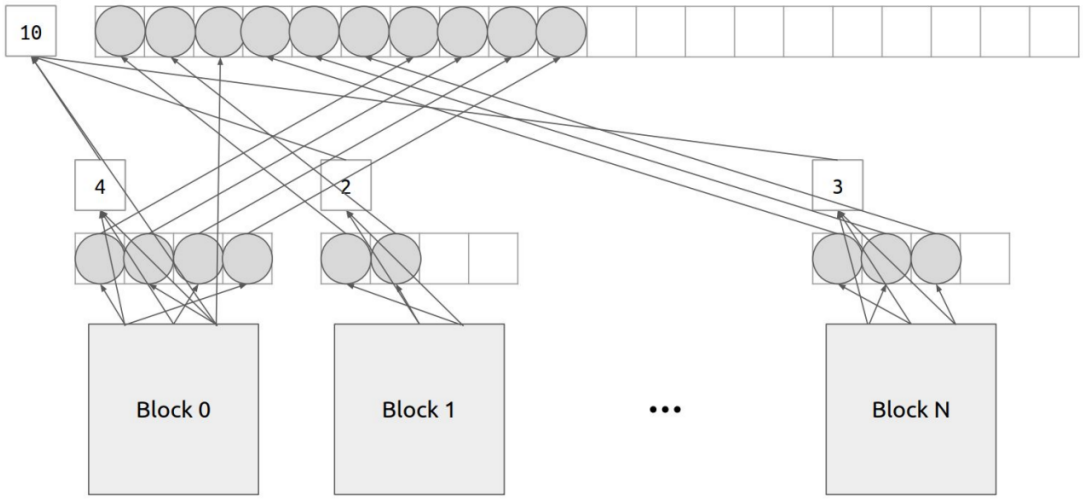
- **Without Synchronization**:
  - Both threads may attempt to visit the same vertex (C), leading to redundancy.
  - This results in two entries for vertex C in the current queue.
- **With `atomicExch`**:
  - The `atomicExch` ensures that only one thread successfully marks vertex C as visited.
  - This prevents redundant entries, leading to a single entry for vertex C in the current queue.

With synchronization both **visitation check** (check if a vertex has been visited and marks it as visited atomically) and **queue indexing** (each thread gets a unique index in the `currentfrontier` array) are managed.

```
if (t < *previousFrontierSize) {
    const int vertex = previousFrontier[t];
    for (int i = rowPointers[vertex]; i < rowPointers[vertex + 1]; ++i) {
        // Check visitation atomically, avoiding redundant expansion
        const int alreadyVisited = atomicExch(&visited[i], 1);
        if (!alreadyVisited) {
            // We're visiting a new vertex: get a spot in line atomically
            const int queueIndex = atomicAdd(&currentFrontierSize, 1);
            distances[destinations[i]] = distances[vertex] + 1;
            // Place the vertex in line
            currentFrontier[queueIndex] = destinations[i];
        }
    }
}
```

15.1.1 Privatization

With privatization, each block maintains its local queue, reducing the contention significantly and allowing more efficient use of atomic operations.




```
__global__ void BFS_Bqueue_kernel(const int *previousFrontier,
                                  const int *previousFrontierSize,
                                  int *currentFrontier,
                                  int *currentFrontierSize,
                                  const int *rowPointers,
                                  const int *destinations,
                                  int *distances,
                                  int *visited) {

  __shared__ int sharedCurrentFrontier[BLOCK_QUEUE_SIZE];
  __shared__ int sharedCurrentFrontierSize, blockGlobalQueueIndex;

  if (threadIdx.x == 0)
    sharedCurrentFrontierSize = 0;
  __syncthreads();
  const int t = threadIdx.x + blockDim.x * blockIdx.x;
  if (t < *previousFrontierSize) {
    const int vertex = previousFrontier[t];
    for (int i = rowPointers[vertex]; i < rowPointers[vertex + 1]; ++i) {
      const int alreadyVisited = atomicExch(&(visited[destinations[i]]), 1);
      if (!alreadyVisited) {
        distances[destinations[i]] = distances[vertex] + 1;
        const int sharedQueueIndex = atomicAdd(&sharedCurrentFrontierSize, 1);
        if (sharedQueueIndex < BLOCK_QUEUE_SIZE) { // there is space in the local queue
          sharedCurrentFrontier[sharedQueueIndex] = destinations[i];
        } else { // go directly to the global queue
          sharedCurrentFrontierSize         = BLOCK_QUEUE_SIZE;
          const int globalQueueIndex        = atomicAdd(currentFrontierSize, 1);
          currentFrontier[globalQueueIndex] = destinations[i];
        }
      }
    }
  }
  __syncthreads();
```

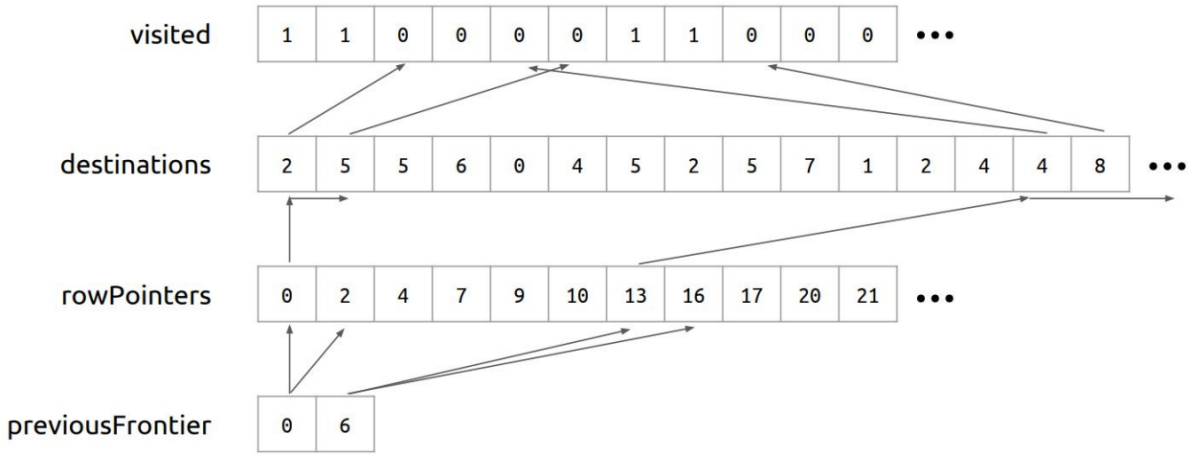
```
  if (threadIdx.x == 0)
    blockGlobalQueueIndex = atomicAdd(currentFrontierSize, sharedCurrentFrontierSize);

  __syncthreads();

  for (int i = threadIdx.x;
  i < sharedCurrentFrontierSize; i += blockDim.x) {

    currentFrontier[blockGlobalQueueIndex + i] = sharedCurrentFrontier[i];
  }
}
```

15.1.2 Texture Memory bitches

Texture memory, which is useful for real-time texture interpolation in graphical applications is also a good fit for this use case because it allows for **efficient unpredictable access** to the graph data.

```
graph TD
    A[Is the usage pattern known, local, and not more than ~48KB?] -->|Yes| B[Shared memory!]
    A -->|No| C[Is read-only memory OK?]
    C -->|Yes| D[Is it less than ~64KB?]
    C -->|No| E[Hope you've been nice to your L2 cache...]
    D -->|Yes| F[Constant memory!]
    D -->|No| G[Texture memory!]
```

The memory access pattern in this scenario is **irregular**.




```
// allocate texture memory
  cudaTextureObject_t rowPointersTexture;
  cudaArray *texArray                = 0;
  cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<int>();
  CHECK(cudaMallocArray(&texArray, &channelDesc, num_vals));
  CHECK(cudaMemcpy2DToArray(texArray,
                            0,
                            0,
```

```
                                rowPointers,
                                sizeof(int) * num_vals,
                                sizeof(int) * num_vals,
                                1,
                                cudaMemcpyHostToDevice));
```

And then:

```
const int t = threadIdx.x + blockDim.x * blockIdx.x;

if (t < *previousFrontierSize) {
    const int vertex = previousFrontier[t];
    for (int i = tex1D<int>(rowPointersTexture, vertex); i < tex1D<int>(rowPointersTexture, vertex + 1);
      // check visitation atomically, avoiding redundant expansion
      const int alreadyVisited =
      atomicExch(&(visited[destinations[i]]), 1);

      if (!alreadyVisited) {
        // we're visiting a new vertex: get a spot in line atomically
        const int queueIndex = atomicAdd(currentFrontierSize, 1);
        distances[destinations[i]] = distances[vertex] + 1;
        // place the vertex in line line
        currentFrontier[queueIndex] = destinations[i];
      }
    }
}
```

- `tex1D<int>(rowPointersTexture, vertex)`: Accesses the row pointer for the given vertex using texture memory. This allows efficient, unpredictable access to the graph data stored in `rowPointersTexture`.
- `tex1D<int>(rowPointersTexture, vertex + 1)`: Accesses the next row pointer to determine the range of outgoing edges for the vertex.

15.1.3 Hybrid GPU-CPU Computation and Memory Management

At the beginning of BFS, the frontier (set of vertices to be explored) is often quite small. The **overhead of launching GPU kernels** can outweigh the benefits of parallel computation in such cases.

As the frontier grows, the parallelism offered by the GPU becomes more advantageous. Therefore, switching between CPU and GPU based on frontier size can optimize performance.

The process involves transferring data between the host (CPU) and the device (GPU). This "ping-pong" effect requires careful management to minimize overhead. Overall:

1) Start on CPU
2) at a certain point, when the frontier size **starts to grow** and remains **stable** (variance on the size must not cause ping-pong) data is transferred on GPU
3) at a certain point, when the frontier **starts to shrink** return to CPU
4) Finish on CPU

16 Brief overview of OpenACC and OpenCL

16.1 OpenACC

OpenACC is a directive-based language used to accelerate code on various architectures, not just GPUs. It simplifies the process compared to CUDA, especially in terms of restructuring code and data management through

concepts like unified memory. The goal is to make code acceleration more efficient and **less time-consuming**:

- **compiler-based** acceleration: annotations to guide the compiler in restructuring the code for acceleration.
- **high portability**: the compiler can target different architectures like GPUs or multicore CPUs just by recompiling. The device and host may be the same physical device in some cases. This allows quick acceleration even for non-skilled programmers, reducing time to market.

While OpenACC provides high programmability through simple annotations of existing code, it may offer lower performance compared to CUDA due to its support for multiple devices and higher level of abstraction. In OpenACC, the parallelism model involves:

- **Gangs**: Corresponding to CUDA blocks, fully independent execution units.
- **Workers**: Each elaborates a vector of work, can synchronize within a gang. **Workers as CUDA Threads**: Each worker executes the same instructions on multiple data.
- **Vectors**: Execute the same instruction on multiple data, similar to SIMD execution. Each vector element is considered a CUDA thread.

Note that each worker could be interpreted as:

- a CUDA thread, executing the same instructions on multiple data
- as a warp, with each vector element corresponding to a CUDA thread

Compiling with OpenACC-capable compiler (for example NVIDIA's nvc compiler) looks like this:

```
# Basic compilation for NVIDIA GPU
nvc -fast -ta=tesla -Minfo=accel -o program_name source_file.c

# -fast: Use all possible optimizations
# -ta=tesla: Target NVIDIA Tesla GPU
# -Minfo=accel: Provide information on how code has been parallelized

# To target a multicore CPU instead of GPU
nvc -fast -ta=multicore -Minfo=accel -o program_name source_file.c

# To use CUDA managed memory (if supported)
nvc -fast -ta=tesla:managed -Minfo=accel -o program_name source_file.c
```

16.1.1 OpenACC Directives

OpenACC directives in C/C++ are specified as pragmas, which are ignored by compilers that don't support them.

```
#pragma acc kernels
for(int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

The `kernels` directive lets the compiler decide on parallelization, while the parallel directive requires the programmer to specify it for each loop.

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

The OpenACC compiler can improve performance by collapsing nested loops and utilizing kernel and `parallel` directives.

```
#pragma acc parallel loop collapse(2)
for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        C[i*N+j] = A[i*N+j] + B[i*N+j];
```

Without `collapse(2)` parallelization might be limited by the outer loop while with it the two loops are collapsed into a single loop and parallelization can be applied across $N \cdot N$ iterations.

`seq` Directive: Executes a loop sequentially within a parallel region. Applying a sort of **coarsening** strategy.

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    #pragma acc loop seq
    for (int j = 0; j < N; j++) {
        C[i*N + j] = A[i*N + j] + B[i*N + j];
    }
}
```

`reduction` Directive: Performs parallel reductions with supported operators `+`, `*`, `max`, `min`, `&`, `|`, `&&`, `||`.

```
int sum = 0;
#pragma acc parallel loop reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
```

`gang(x)` clause: divides a loop into `x` independent gangs for parallel execution. Each gang can run independently on different processing units. `worker` then optionally used to specify how to parallelize loops.

```
#pragma acc parallel loop gang(16)
for (int i = 0; i < N; i++) {
    #pragma acc loop worker
    for (int j = 0; j < N; j++) {
        C[i*N + j] = A[i*N + j] + B[i*N + j];
    }
}
```

Except for advanced cases, the organization of gangs, workers, and vectors can be left entirely to the compiler. Proper handling of private variables with clauses like `private` is essential to avoid issues of global access.

```
#pragma acc parallel loop private(swap)
for(int i = 0; i < N; i++){
    int swap = A[i];
    A[i] = B[i];
    B[i] = swap;
}
```

Different levels of parallelism can be specified for each loop using directives like `gang`. Commands like `copyin`, `copyout`, and `create` handle data transfer tasks:

- **`copyin`**: Data are copied from the host to the device at the beginning of the parallel region, and memory is freed at the end of the region.
- **`copyout`**: Device memory is allocated at the beginning of the parallel region and copied to the host memory at the end of the region.
- **`copy`**: Combines `copyin` and `copyout` behaviors.
- **`create`**: Allocates device memory.

The compiler then decides how and when to move data between the host and the device.

```
#pragma acc data copyin(A[0:N], B[0:N]) copyout(C[0:N])
{
    #pragma acc kernels
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

16.1.2 Example of Jacobi iterative method

Jacobi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface.

```
#pragma acc data create(Anew[0:n][0:m]) copy(A[0:n][0:m])
while (err > tol && iter < iter_max) {
    err = 0.0;
    #pragma acc parallel loop reduction(max:err) collapse(2)
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
        //temperature of each single point is computed as
        // linear comb of 4 neighbours
            Anew[j][i] = 0.25 * (A[j][i+1]
            + A[j][i-1]
            + A[j-1][i]
            + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    #pragma acc parallel loop collapse(2)
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

16.1.3 Parallelization Considerations for Loops

- **While loop**: Has data dependencies among iterations and cannot be parallelized.
- **Nested for loops**: Iterations are independent and can be parallelized so `collapse(2)`
- **Error computation**: `err` is computed using a reduction clause.
- **Data transfer**:
  - `A`: Transferred to the device at the beginning and back to the host at the end. `copy(A[0:n][0:m])`
  - `Anew`: Used only on the device, so `create(Anew[0:n][0:m])`

16.2 OpenCL

OpenCL acts as a runtime that enables the programming of diverse computing resources using a single language. While CUDA is designed by NVIDIA and it's tailored to **specific** accelerators while OpenCL is a comprehensive solution which:

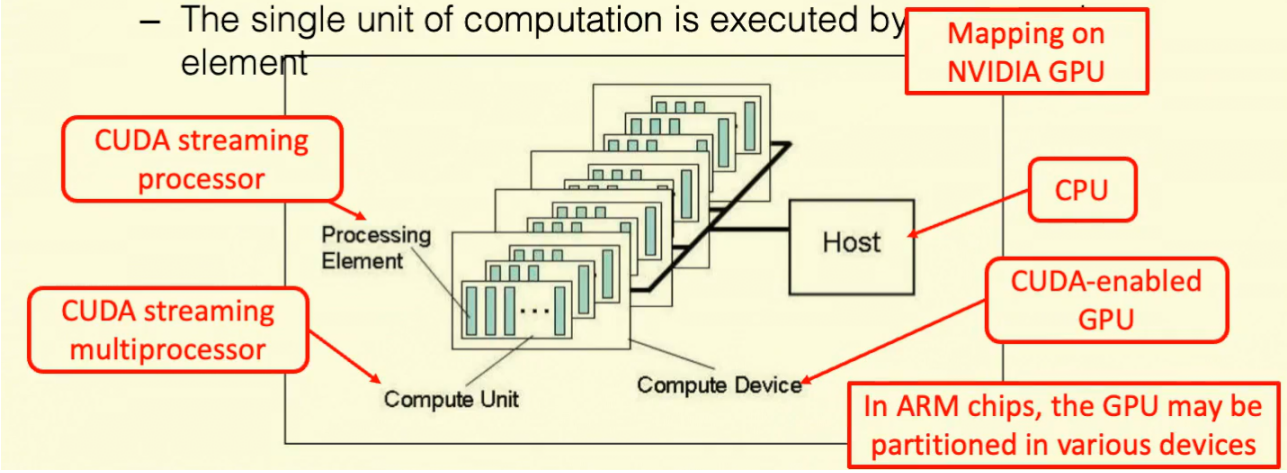
- supports different types of accelerators beyond just GPUs, including those found in desktops, HPC systems, service machines, mobile devices, and **even FPGAs**.
- optimizes performance while considering energy consumption, especially in mobile markets where efficiency and trade-offs between compute units like CPUs and GPUs are crucial.

The development of OpenCL was spearheaded by Apple and is now managed by the Khronos Group, a consortium of hardware and software companies.

OpenCL supported Parallelism Models:

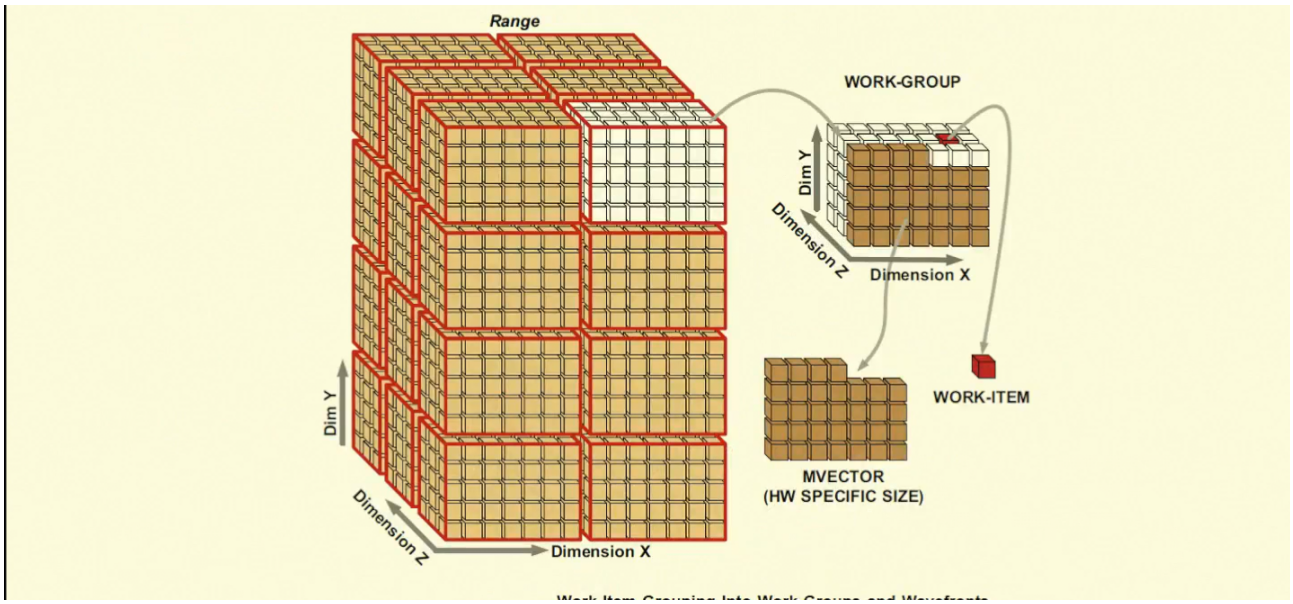
- **Single-Instruction-Multiple-Data (SIMD)**
  - The kernel is composed of sequential instructions.
  - The instruction stream is executed in lock-step on all involved processing elements.
  - Generally used on GPU.
- **Single-Program-Multiple-Data (SPMD)**
  - The kernel contains loops and conditional instructions.
  - Each processing element has its own program counter.
  - Each instance of the kernel has a different execution flow.
  - Generally used on CPU.
- **Data-Parallel Programming Model**
  - The same sequential instruction stream is executed in lock-step on different data.
  - Generally used on GPU.
- **Task-Parallel Programming Model**
  - The program issues many kernels in an out-of-order fashion.

The platform model involves multiple compute devices, each comprising compute units and processing elements for parallel processing. OpenCL's program structure mirrors CUDA, with host code written in C++ for sequential tasks and device code in OpenCL C for managing kernels and data parallel execution.

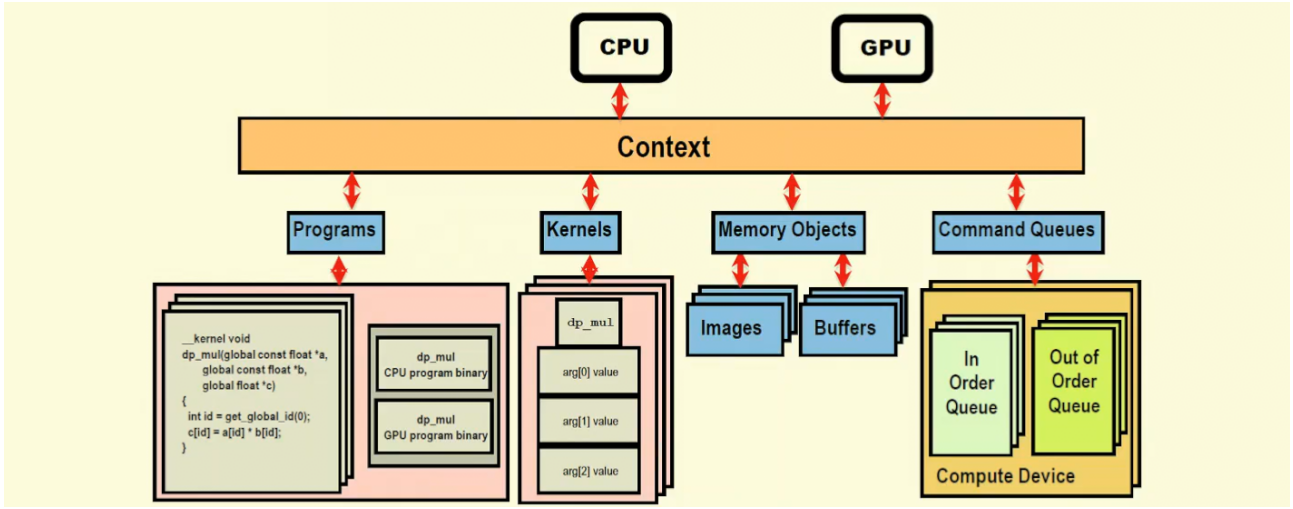


OpenCL's memory model is comparable to CUDA's, including private, local, global, and constant memory types.

OpenCL Term	CUDA Equivalent	Description
NDrange	CUDA grid	N-dimensional grid with a local and global indexing, spanned by work-groups
Work-group	CUDA block	Group of contiguous work-items
Work-item	CUDA thread	Single execution of the kernel on a data instance (i.e., the basic unit of work)



The program object encapsulates both source and compiled code, which is compiled at runtime based on the selected device, using just-in-time compilation. Memory objects like buffers and images are employed for data transmission.



The context object encapsulates various features like command queues, unlike CUDA, where much is managed by the runtime due to its focus on a single accelerator

Just-in-time compilation in OpenCL provides flexibility across different devices, compiling device-side code at runtime using LLVM.

16.2.1 Code and directives

Very very similar to CUDA to advantage engineers to easily switch between the 2 languages. OpenCL kernels are implemented in OpenCL C, an extension of C99 with some restrictions:

- No function pointers
- No recursion
- No variable-length arrays

New data types include:

- Scalar types: `half`, ...
- Image types: `image2d_t`, `image3d_t`, `sampler_t`
- Vector types: `char2`, `ushort4`, `int8`, `float16`, `double2`, ...

Address space qualifiers:

- `__global`, `__local`, `__constant`, `__private`

Built-in functions:

- Work-item functions: `get_global_id`, `get_local_id`, ...
- Math functions: `sin`, `cos`, `exp`, ...
- Common functions: `min`, `clamp`, `max`, ...
- Geometric functions: `cross`, `dot`, `normalize`, ...
- Vector load/store functions: `vload4`, `vstore4`, ...
- Synchronization functions: `barrier`
- Atomic functions: `atomic_add`, `atomic_sub`, ...

16.3 Example OpenCL kernel

```
__kernel void dp_mult(__global const float* a,
                      __global const float* b,
                      __global float* c)
{
    int i = get_global_id(0);
    c[i] = a[i] * b[i];
}
```

16.4 Example with problem size parameter

```
__kernel void dp_mult(__global float* a,
                      __global float* b,
                      __global float* c,
                      int N)
{
    int i = get_global_id(0);
    if(i < N)
        c[i] = a[i] * b[i];
}
```

16.5 Matrix multiplication example

```
__kernel void parMultiply1 (
    __global float *A, __global float *B,
    __global float *C)
{
    // Vector element index
    const int m = get_global_id(0);
    const int n = get_global_id(1);
    const int M = get_global_size(0);

    C[m + M * n] = 0;
    for (int k = 0; k < M; k++)
```

```
        C[m+M*n] += A[m+M*k] * B[k+M*n];
}
```

16.6 Matrix multiplication with local memory

```
__kernel void parMultiply2 (__global float *A, __global float *B,
                            __global float *C, int K)
{
    const int row = get_local_id(0); // Local row ID (max: TS)
    const int col = get_local_id(1); // Local col ID (max: TS)
    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
    const int M = get_global_size(0);
    const int N = get_global_size(1);

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    float acc = 0.0f;
    const int numTiles = K/TS;
    for (int t=0; t<numTiles; t++) {
        const int tiledRow = TS*t + row;
        const int tiledCol = TS*t + col;
        Asub[col][row] = A[tiledCol*M + globalRow];
        Bsub[col][row] = B[globalCol*K + tiledRow];

        barrier(CLK_LOCAL_MEM_FENCE);

        for (int k=0; k<TS; k++) {
            acc += Asub[k][row] * Bsub[col][k];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    C[globalCol*M + globalRow] = acc;
}
```

16.6.1 Workflow of an OpenCL Application

The host code in OpenCL is more elaborate than in CUDA:

1. **Discovering the Platforms and Devices**
2. **Creating a Context**
3. **Creating a Command Queue per Device**
4. **Creating Memory Objects to Hold Data**
5. **Copying the Input Data onto the Device**
6. **Creating and Compiling a Program from the OpenCL C Source Code**
7. **Generating a Kernel of the Program and Specifying Parameters**
8. **Executing the Kernel**
9. **Copying Output Data Back to the Host**
10. **Releasing the OpenCL Resources**

Here's a more detailed breakdown of the workflow:

1. **Discovering the Platforms and Devices**
   - Platforms correspond to vendor-specific libraries
   - Use functions like `clGetPlatformID` and `clGetPlatformInfo` to identify available platforms
   - Each platform can have multiple devices
   - Use `clGetDeviceID` and `clGetDeviceInfo` to select device IDs for acceleration tasks

```
cl_uint numPlatforms;
cl_platform_id *platformIds;
clGetPlatformIDs(0, NULL, &numPlatforms);
platformIds = (cl_platform_id *)malloc(sizeof(cl_platform_id) * numPlatforms);
clGetPlatformIDs(numPlatforms, platformIds, NULL);

cl_uint numDevices;
cl_device_id *deviceIds;
clGetDeviceIDs(platformIds[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
deviceIds = (cl_device_id *)malloc(sizeof(cl_device_id) * numDevices);
clGetDeviceIDs(platformIds[0], CL_DEVICE_TYPE_ALL, numDevices, deviceIds, NULL);
```

2. **Creating a Context**
   - A context is a container for associated devices, program objects, kernels, memory objects, and command queues

```
cl_context context;
cl_context_properties properties[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)platformIds[0], 0}
context = clCreateContext(properties, 1, &deviceIds[0], NULL, NULL, NULL);
```

3. **Creating a Command Queue per Device**
   - Command queues are used for issuing commands to a device

```
cl_command_queue queue;
queue = clCreateCommandQueue(context, deviceIds[0], 0, NULL);
```

4. **Creating Memory Objects to Hold Data**
   - Memory objects are used to transmit data to/from a device

```
cl_mem bufA, bufB, bufC;
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*NUM_DATA, NULL, NULL);
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*NUM_DATA, NULL, NULL);
bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float)*NUM_DATA, NULL, NULL);
```

5. **Copying the Input Data onto the Device**
   - Transfer data from host memory to device memory

```
clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0, sizeof(float)*NUM_DATA, a, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, bufB, CL_TRUE, 0, sizeof(float)*NUM_DATA, b, 0, NULL, NULL);
```

6. **Creating and Compiling a Program from the OpenCL C Source Code**
   - Load and compile the kernel source code

```
const char *programSource = "__kernel void example(...) { ... }";
cl_program program = clCreateProgramWithSource(context, 1, &programSource, NULL, NULL);
clBuildProgram(program, 1, &deviceIds[0], NULL, NULL, NULL);
```

7. **Generating a Kernel of the Program and Specifying Parameters**
   - Create a kernel from the compiled program and set its arguments

```
cl_kernel kernel = clCreateKernel(program, "example", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```

8. **Executing the Kernel**
   - Launch the kernel execution on the device

```
size_t globalWorkSize[1] = { NUM_DATA };
```

```
size_t localWorkSize[1] = { 64 };
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
```

9. **Copying Output Data Back to the Host**
   - Transfer results from device memory back to host memory

```
float results[NUM_DATA];
clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0, sizeof(float)*NUM_DATA, results, 0, NULL, NULL);
```

10. **Releasing the OpenCL Resources**
    - Free all allocated OpenCL resources

```
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

17 HSA foundations

The Heterogeneous System Architecture (HSA) represents a paradigm shift in heterogeneous computing, aiming to seamlessly integrate CPUs, GPUs, and other accelerators. Founded in June 2012:

- Developing a new platform for heterogeneous systems
- Aims to unite accelerators architecturally
- Initial focus on GPU compute, but expanding beyond

Goals:

- Enable power-efficient performance
- Improve programmability of heterogeneous processors
- Increase portability of code across processors and platforms
- Increase pervasiveness of heterogeneous solutions

17.1 Key Features

Built on three key pillars:

- hUMA
- hQ
- HSAIL

17.1.1 hUMA (Heterogeneous Unified Memory Architecture)

- Unified coherent memory enables data sharing across all processors
- Allows usage of pointers
- No explicit data transfer - values move on demand
- Pageable virtual addresses for GPUs - no GPU capacity constraints
- CPU and GPU have unified virtual memory spaces

17.1.2 hQ (Heterogeneous Queuing)

- User mode queuing for low latency dispatch
- Architected Queuing Layer (AQL) enables any agent to enqueue tasks
- Single compute dispatch path for all hardware
- No driver translation, direct access to hardware

- Allows dispatch to queue from any agent (CPU or GPU)
- GPU self-enqueue enables solutions like recursion and tree traversal

17.1.3 HSAIL (HSA Intermediate Language)

- Portable "virtual ISA" for vendor-independent compilation and distribution
- Low-level IR, close to machine ISA level
- Generated by high-level compilers (LLVM, gcc, Java VM, etc.)
- Compiled to target ISA by vendor-specific "finalizer"

17.2 Advantages over Legacy GPU Compute

- Eliminates multiple memory pools and address spaces
- No explicit data copying across PCIe
- Lower dispatch overhead
- No need for lots of compute on GPU to amortize copy overhead
- Removes GPU memory capacity limitations
- Eliminates dual source development
- More accessible to non-expert programmers
- Enables natural expression of nested parallelism
- Removes synchronization and communication overhead with the host
- Exposes finer granularities of parallelism to scheduler and load balancer
- Task preemption and context switching support on all computing resources (including GPUs)

17.3 AMD Carrizo

As one of the founding members of the HSA Foundation in 2012, **AMD** was instrumental in driving the initiative forward.

AMD Carrizo, launched in 2015, was one of the first processors to fully implement HSA 1.0 specifications. It was part of AMD's APU (Accelerated Processing Unit) lineup, which combined CPU and GPU cores on a single chip.

Carrizo represented a significant step forward in realizing the HSA vision, offering:

1. Full support for heterogeneous unified memory architecture (hUMA)
2. Hardware-based GPU scheduling
3. User-mode queuing
4. Shared virtual memory between CPU and GPU

While HSA as a specific standard hasn't materialized as initially hoped, many of its core ideas have influenced the broader industry: AMD with its APU designs, Apple's M1/M2 chips, NVIDIA's unified memory in CUDA and Intel's integrated GPUs incorporate many HSA principles.