Parallel Programming and Heterogeneous Computing

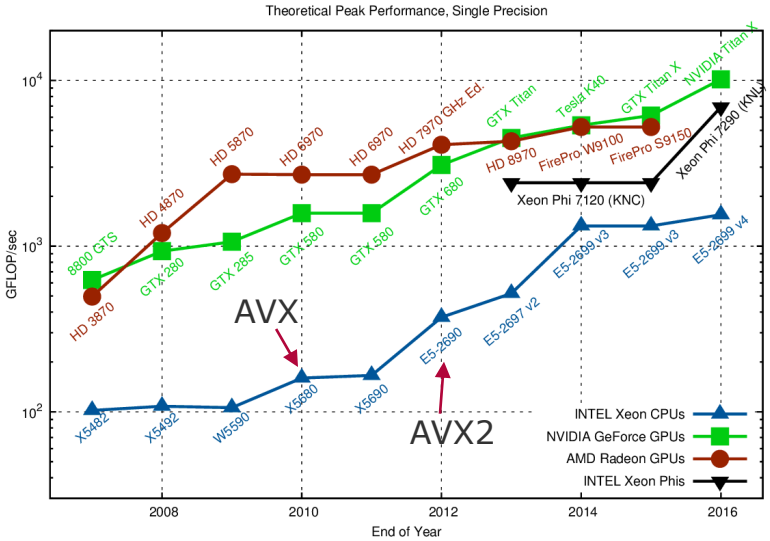
Heterogeneous Computing with GPUs and CUDA

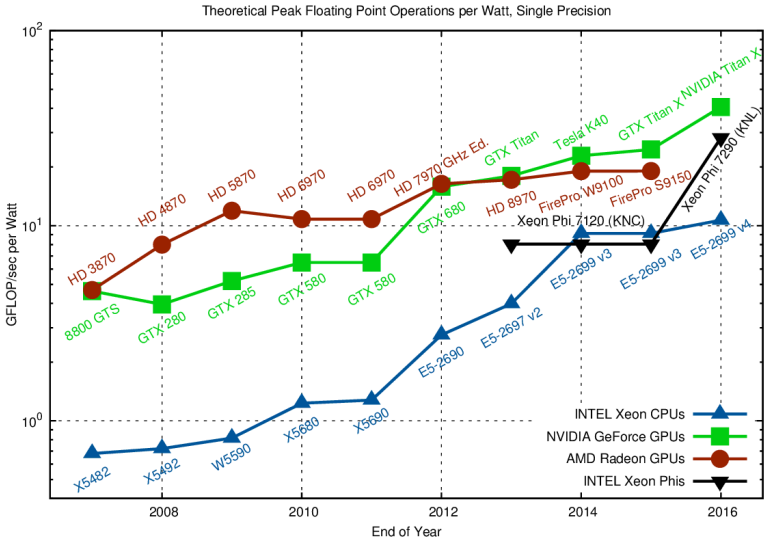
Max Plauth, Sven Köhler, Felix Eberhardt, Lukas Wenzel and Andreas Polze

Operating Systems and Middleware Group

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Why GPUs?





AVX-512

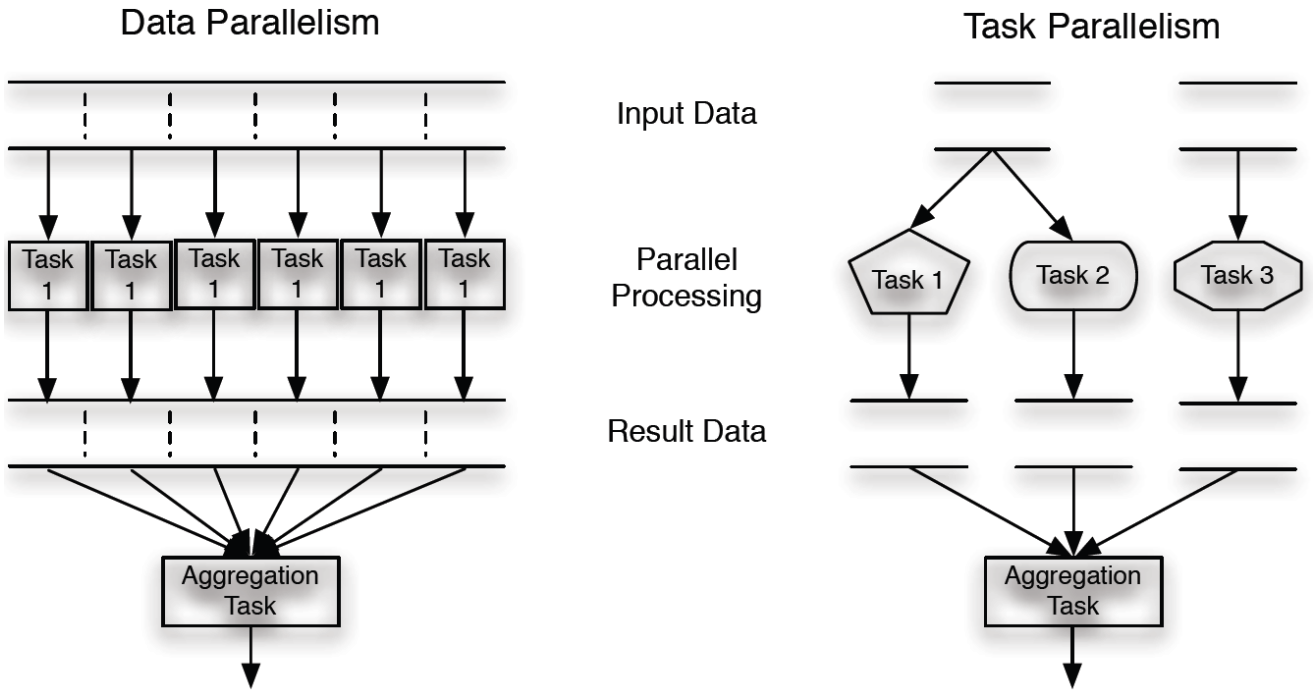
- >25% of HPC systems in the Top500 (Nov ’18) are powered by GPUs

[https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, https://www.top500.org/statistics/list/]

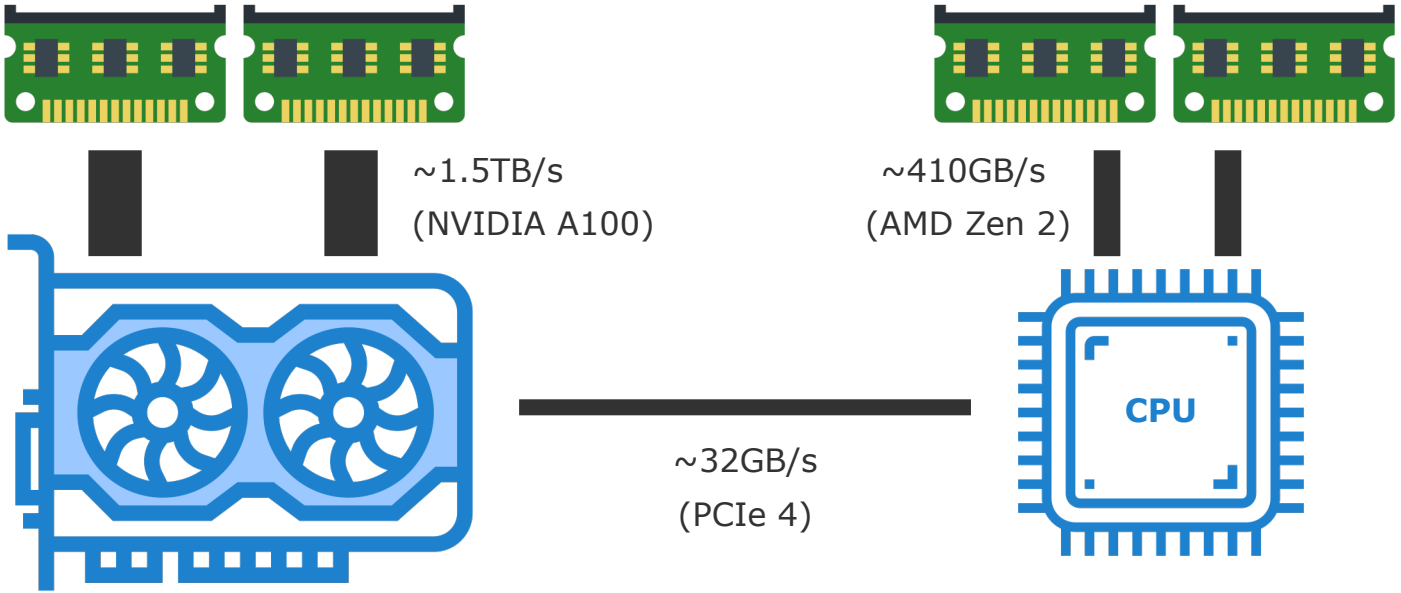
A Brief History of GPUs

Fixed Function Graphic Pipelines	•1980s-1990s; configurable, not programmable; first APIs (DirectX, OpenGL); Vertex Processing
Programmable Real-Time Graphics	•Since 2001: APIs for Vertex Shading, Pixel Shading and texture manipulation; DirectX9
Unified Graphics and Computing Processors	•2006: NVIDIAs G80; unified processors arrays; three programmable shading stages; DirectX10
General Purpose GPU (GPGPU)	•Compute problem as native graphic operations; algorithms as shaders; data in textures
GPU Computing	•CUDA (2007) / OpenCL (2009); programmable shaders; load and store instructions; barriers; atomics

Recap: Data vs. Task Parallelism

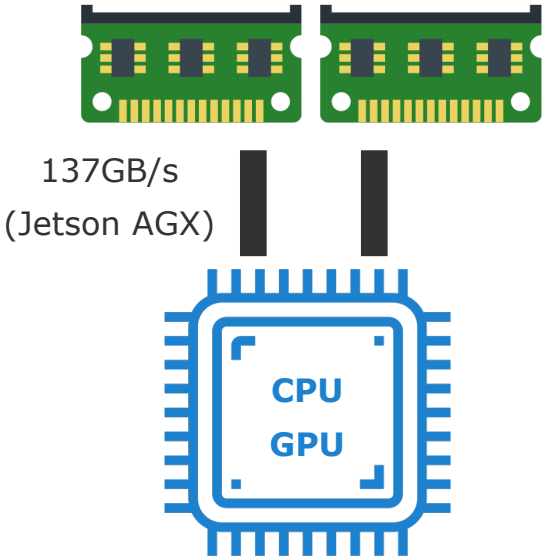


GPU Hardware: Discrete GPUs



[Icons by srip, monkik, and Smashicons from www.flaticon.com]

GPU Hardware: Integrated GPUs



[Icons by srip, monkik, and Smashicons from www.flaticon.com]

Hardware: NVIDIA GA100 Full GPU with 128 SMs

ParProg20 C2 GPUs

Max Plauth

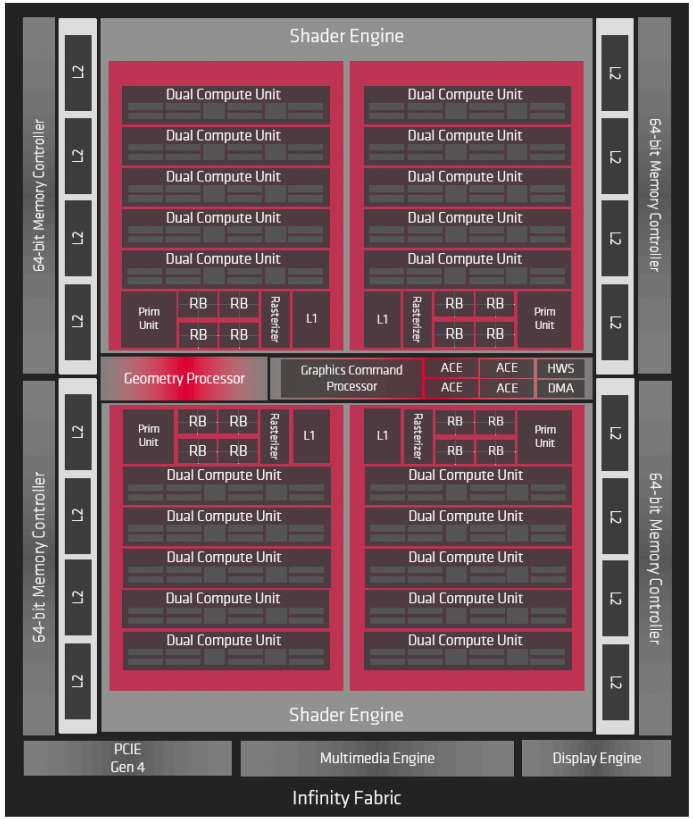
Hardware: NVIDIA GA100 SM

ParProg20 C2 GPUs

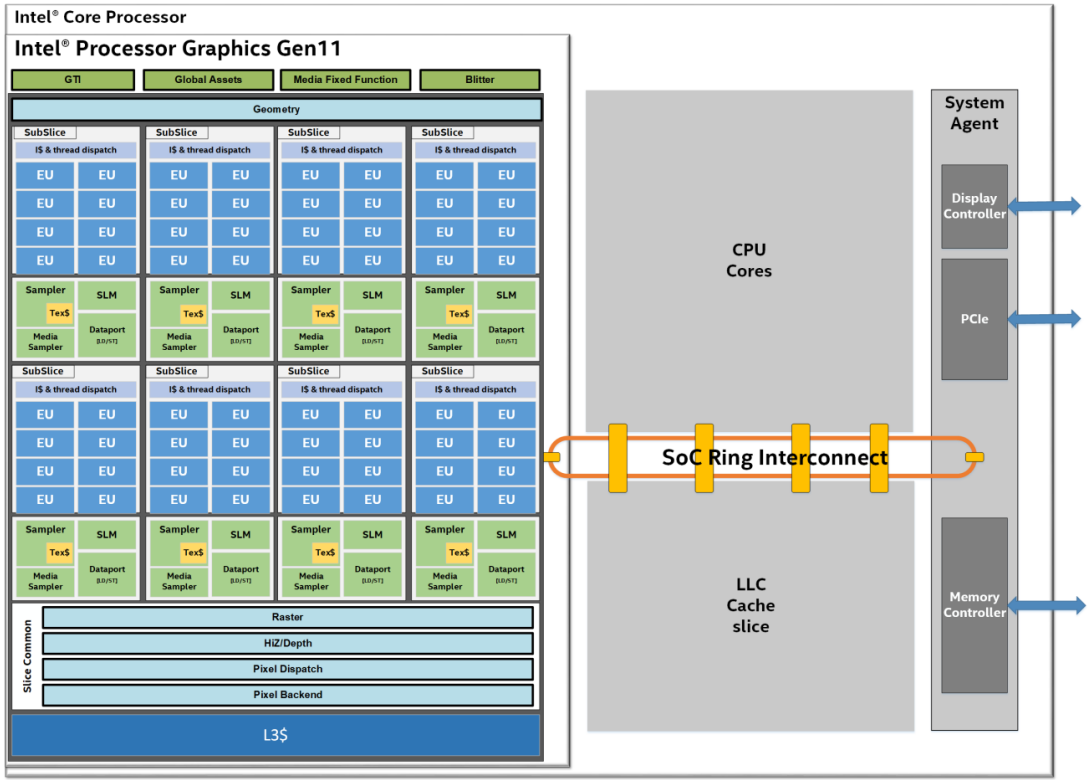
Max Plauth

Chart **8**

Hardware: AMD RDNA/Navi GPU



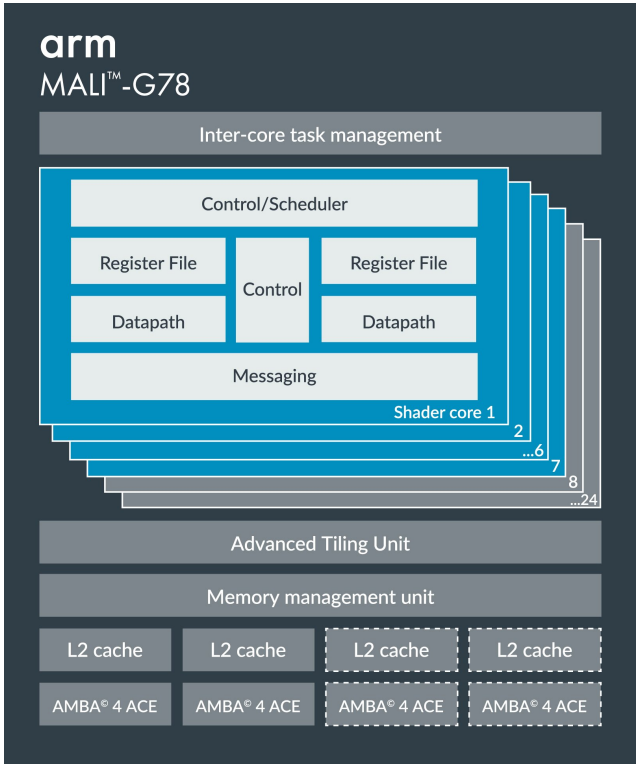
Hardware: Intel Iris Gen11 iGPU



ParProg20 C2 GPUs

Max Plauth

Hardware: ARM Mali 2nd Gen Valhall iGPU

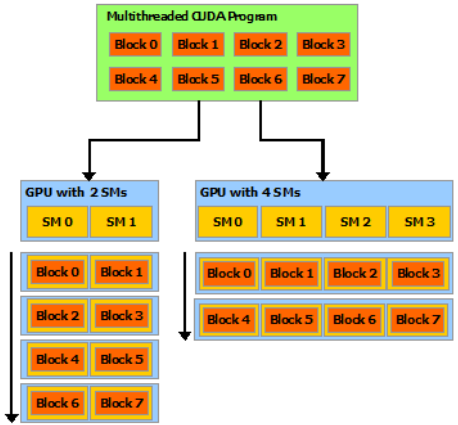


(Vendor independent) properties of GPUs

- Current GPU designs are based on many superscalar "cores"
- "Cores" are grouped in SMs/Compute Engines/Subslices/Shader Cores/...
- Unlike CPUs, GPU "cores" cannot operate independently from each other
  - "Cores" share control-flow logic and operate in warps / wavefronts
  - Number of "cores" per warp / wavefront varies from vendor to vendor
  - Branching within a warp results in serialized execution → expensive
- Memory bandwidth increases, but latency can hardly be improved
  - Large register file / L1$ required to support many active threads
  - #active threads >> #cores required for latency hiding
- Complex memory hierarchy must be managed by software explicitly

CUDA Programming Model

- Three key abstractions
  - A hierarchy of threads groups
  - Shared memories
  - Barrier Synchronization



- "Foster says hello!"
  - Partitioning/Communication:
    - Each thread performs smallest possible task
  - Agglomeration/Mapping:
    - Coarse tasks are performed by blocks of threads
    - Blocks enable scalability from entry level to enthusiast level GPUs

CUDA Programming Model: Kernels

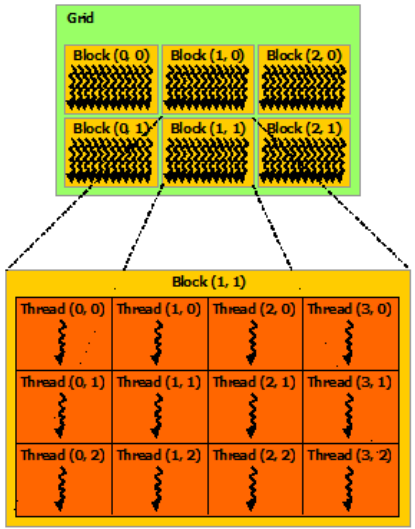
- „a routine compiled for high throughput accelerators“ (Wikipedia)
- An instance of a kernel function is executed once per thread
- Indices determine what portion of work is performed by a kernel instance
- Think of kernels as the body of an inner loop

```
void
serial_mul(const float* a,
           const float* b,
           float* c,
           int n)
{
  for(int i = 0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

```
__global__ void
mul(__global__ const float* a,
    __global__ const float* b,
    __global__       float* c)
{
  int id = threadIdx.x +
     blockIdx.x * blockDim.x;
  c[id] = a[id] * b[id];
}
```

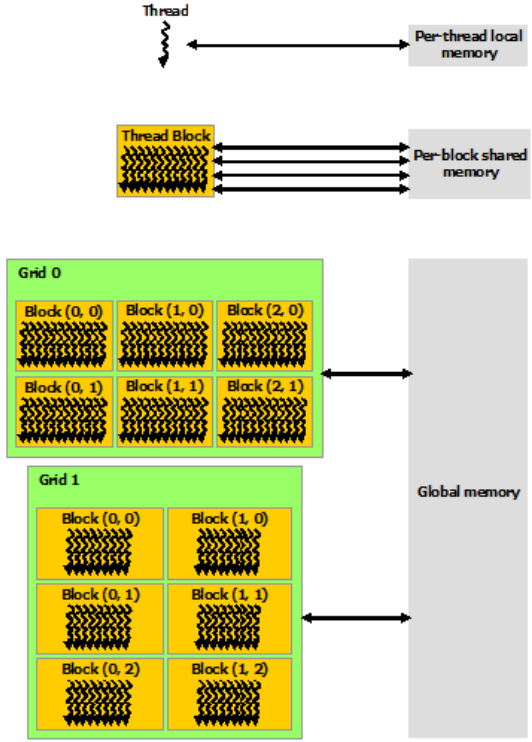
CUDA Programming Model: Thread Hierarchy

- Each thread only performs light work
  - e.g. performs an operation on a single array element
- Threads are grouped in blocks
  - Threads are identified using the built-in, 3-component index vector `threadIdx`
  - The dimensions of the thread blocks are accessible via the `blockDim` vector
- All blocks are organized in the grid
  - Blocks are identified using the built-in, 3-component index vector `blockIdx`



CUDA Programming Model: Memory Hierarchy

- Register File
  - Private to each thread
  - Fastest memory, several variables
- Shared Memory
  - Shared per block
  - Fast memory, several kilobytes
  - Managed manually
- Global Memory
  - Shared per process
  - Slowest memory, several gigabytes



CUDA Execution Model

- Single Instruction, Multiple Threads
  - Each thread executes the same code
- A thread block is executed on one streaming multiprocessor (SM)
  - Synchronization and communication is only possible within blocks
  - Inter-block data exchange is only possible via global memory
- Warps: a SM schedules/executes threads in units of 32 threads
  - Warps (used to) share a single program counter amongst all 32 threads
  - Divergent code results in serialized execution
  - Synchronization in divergent code will lead to deadlocks

CUDA C++

- Single-Source Approach:
  - Host and device code can be mixed in the same source files
- CUDA 10: superset of ISO C++14 with additions including
  - Function Execution Space Specifiers
    - `__global__`, `__device__`, `__host__`, `__noinline__`, `__forceinline__`
  - Variable Memory Space Specifiers
    - `__device__`, `__constant__`, `__shared__`, `__managed__`, `__restrict__`
  - Built-in Variables
    - `threadIdx`, `blockDim`, `blockIdx`, `gridDim`, `warpSize`
  - Synchronization Functions
    - `cudaDeviceSynchronize()`, `cudaStreamSynchronize()`, …
  - Memory Management
    - `cudaMalloc()`, `cudaFree()`, `cudaMallocHost()`, `cudaFreeHost()` …

Simple CUDA Code Workflow

1. Allocate host memory (and prepare input data)
2. Allocate device memory
3. Copy input data from host memory to device memory
4. Launch Kernel
5. Copy result from device memory to host memory
6. Free resources

Example: Vector Addition CUDA Kernel

- Kernel body is instantiated once for each thread
  - Each thread has a unique index

```
__global__ void vectorAdd(__global__ const float *a,
                          __global__ const float *b,
                          __global__       float *c)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    c[i] = a[i] + b[i];

}
```

Code that actually executes on the GPU

Example: Vector Addition Host Code

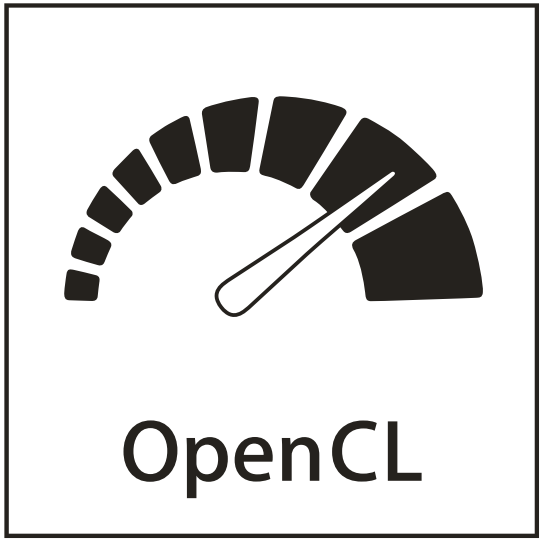
```
// Error code to check return values for CUDA calls
cudaError_t err = cudaSuccess;

// Print the vector length to be used, and compute its size
int numElements = 50000;
size_t size = numElements * sizeof(float);

// Allocate the host input vector A, B, and C
float *h_A = (float *)malloc(size);
float *h_B = (float *)malloc(size);
float *h_C = (float *)malloc(size);

// Initialize the host input vectors
for (int i = 0; i < numElements; ++i)
{
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
}

// Allocate the device input vector A
float *d_A = NULL;
float *d_B = NULL;
float *d_C = NULL;
err = cudaMalloc((void **)&d_A, size);
err = cudaMalloc((void **)&d_A, size);
err = cudaMalloc((void **)&d_A, size);
```

```
// Copy the host input vectors A and B to device memory
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) /
threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
numElements);
err = cudaGetLastError();

// Copy the device result back to host memory.
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device global memory
err = cudaFree(d_A);
err = cudaFree(d_B);
err = cudaFree(d_C);

// Free host memory
free(h_A);
free(h_B);
free(h_C);

return 0;
```

A brief comparison: CUDA vs. OpenCL

CUDA®

- Vendor-dependent
- Exposes hardware features
- GPUs only



- Vendor-independent standard
- No direct access to hardware
- CPUs, GPUs, DSPs, FPGAs...

Terminology: CUDA vs. OpenCL

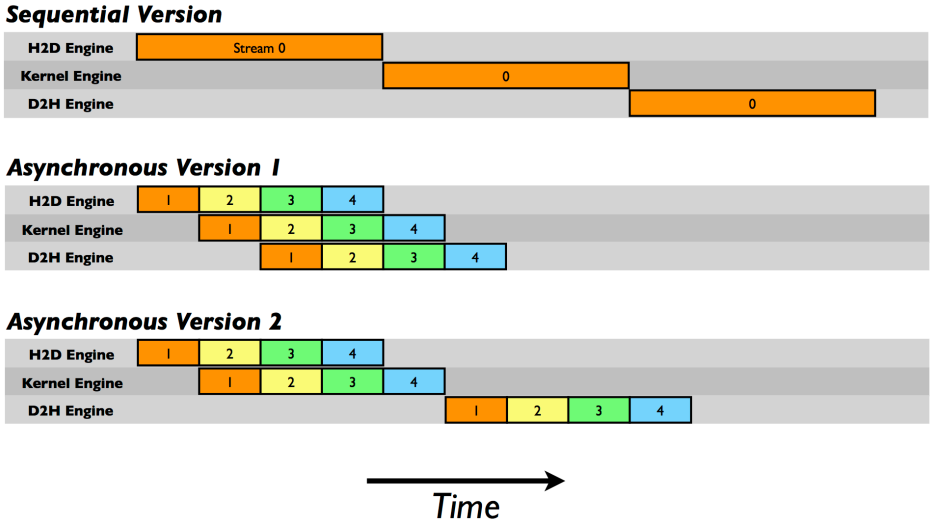
Term	CUDA	OpenCL
	grid	NDRange
	block	work-group
	thread	work-item
	warp	sub-group
Thread index	threadIdx.x	get_local_id(0)
Group index	blockIdx.x	get_group_id(0)
Group dimension	blockDim.x	get_local_size(0)
Thread count	gridDim.x	get_global_size(0)
Kernel Launch	<<< >>>	clEnqueueNDRangeKernel
Global Memory	__global__	__global
Group Memory	__shared__	__local
Thread Local Storage	__local__	__private
Constant Memory	__constant__	__constant

Best Practices for Performance Tuning

Algorithm Design	• Asynchronous, Recompute, Simple
Memory Transfer	• Chaining, Overlap Transfer & Compute
Control Flow	• Avoid Divergent Branching
Memory Types	• Local Memory as Cache, rare resource
Memory Access	• Coalescing, Bank Conflicts
Sizing	• Work-Group Size, Work / Work-Item
Instructions	• Shifting, Fused Multiply, Vector Types
Precision	• Native Math Functions, Build Options

Performance Tuning: Overlap Transfer & Compute

- Per default, all commands are performed in-order in the default stream
- Level of concurrency can be increased by using multiple streams, e.g. for overlapping memory transfers with computation



Performance Tuning: Overlap Transfer & Compute

- Streams must be created explicitly
  - `cudaCreateStream()`, `cudaDestroyStream()`
- Asynchronous API functions must be used, with stream as parameter
  - Many functions of the CUDA API exist in sync. and async. versions
  - `cudaMemcpy()` vs. `cudaMemcpyAsync()`
- Kernel launches are always asynchronous
  - Explicit synchronization: `cudaDeviceSynchronize()` or `cudaStreamSynchronize()`
  - Implicit synchronization at next sync. call in the same stream, e.g. `cudaMemcpy()`

Performance Tuning: Divergent Branching and Predication

Divergent Branching

- Flow control instruction (`if`, `switch`, `do`, `for`, `while`) can result in different execution paths
- Data parallel execution → varying execution paths will be serialized
- Threads converge back to same execution path after completion

Branch Predication

- Instructions are associated with a per-thread condition code (predicate)
  - All instructions are scheduled for execution
  - Predicate true: executed normally
  - Predicate false: do not write results, do not evaluate addresses, do not read operands
- Compiler may use branch predication for `if` or `switch` statements

Performance Tuning: Shared, Texture, and Constant Memory

Shared Memory

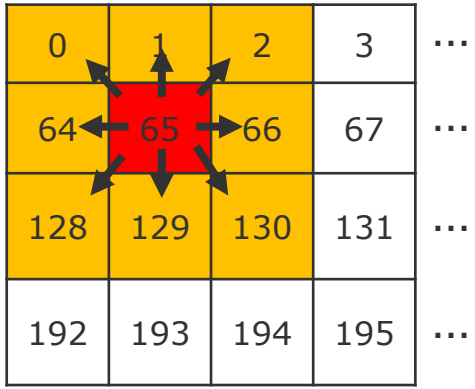
- Memory latency much lower than global memory latency
- Small, no coalescing problems, prone to memory bank conflicts

Texture Memory

- 2-dimensionally cached, read-only
- Can be used to avoid uncoalesced loads form global memory

Constant Memory

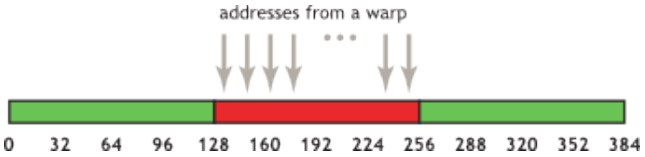
- Cached, read-only, 64 KB



Performance Tuning: Memory Coalescing

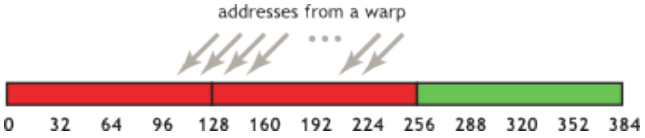
Simple Access Pattern

- The k-th thread accesses the k-th word in a cache line. Not all threads need to participate.
- A single 128B L1 cache line is sufficient.



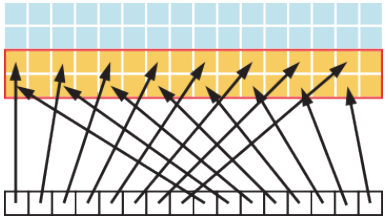
Sequential but Misaligned Access

- Sequential threads access memory that is sequential but not aligned with the cache lines, two 128-byte L1 cache lines will be requested.



Strided Accesses

- Stride of 2 results in a 50% of Ld/St efficiency
- Worst case: One word per cache line is used



Performance Tuning: Memory Coalescing

- Sequential but Misaligned Access

```
__global__ void offset(__global float *a,
                       const int offset) {
    int id = blockDim.x * blockIdx.x +
             threadIdx.x + offset;
    a[id] += 1;
}
```

- Strided Accesses

```
__global__ void stride(__global float *a,
                       const int stride) {
    int id = (blockDim.x * blockIdx.x +
             threadIdx.x) * stride;
    a[id] += 1;
}
```

Heterogeneous Computing with GPUs and CUDA References

- **„CUDA Toolkit Documentation"**. NVIDIA Corporation https://docs.nvidia.com/cuda/
- **„A history of the NVIDIA Stream Multiprocessor"**. Fabien Sanglard, https://fabiensanglard.net/cuda/index.html

- Code Examples for Optimization Techniques
  - **„Using Shared Memory in CUDA C/C++"**. Mark Harris, https://devblogs.nvidia.com/using-shared-memory-cuda-cc/
  - **„How to Access Global Memory Efficiently in CUDA C/C++ Kernels"**. Mark Harris, https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/
  - **„How to Overlap Data Transfers in CUDA C/C++"**. Mark Harris, https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/


Thank you
for your attention!