Back to the Standard

From CUDA and Pragmas to GPU-Accelerated Parallel C++

ELMAR WESTPHAL

Cppcon
The C++ Conference
2025
September 13 - 19

DISCLAIMER

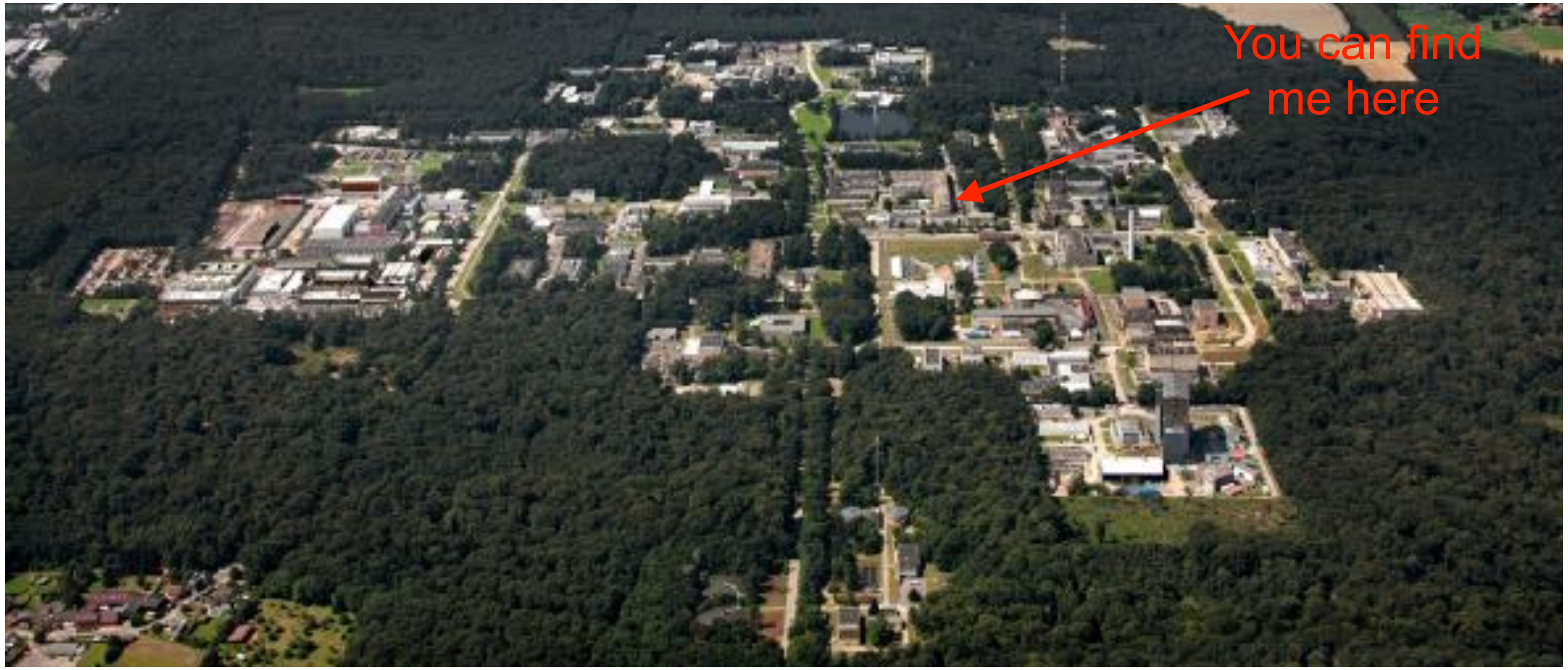
Accelerated Standard C++ is not some kind of magic bullet to be fired at legacy code.

OUTLINE

- Introduction
- GPU Basics
- A sample task used to illustrate different ways of accelerating code on GPUs
- A trip down memory lane: The evolution of GPU memory access models
- Parallel C++ algorithms on GPU
  - How the available memory model influences the seamlessness of integration
  - Supported algorithms
  - Limitations
  - Logic and performance pitfalls

INTRODUCTION

Elmar Westphal, research associate at Forschungszentrum Jülich, PGI/JCNS-TA*, Scientific IT-Systems



*Peter Grünberg Institute/Jülich Centre for Neutron Science, Technical and Administrative Infrastructure

WHY TRUST ME?

- I'm not trying to sell you anything
  - I'm not even affiliated with any of the companies mentioned (it's a different PGI)
- $\sim 10000_2$ years of experience in porting and developing scientific code for GPUs
  - In different domains:
    - Micromagnetism
    - Molecular Dynamics / Hydrodynamic Interactions
    - Neutron Scattering
  - At different scales:
    - From desktop to supercomputer

DOES THE STUFF PRESENTED WORK ON YOUR GPU?

- I usually develop for and work with Nvidia-based systems
- Most things shown also apply to AMD GPUs, at least to some extend
  - I have limited access to and experience with the AMD ecosystem, information is given to the best of my knowledge
- The content of this talk is not applicable to Apple GPUs
  - They may have (deprecated) support for OpenCL
  - There is no support for parallel algorithms for neither CPU or GPU yet
    - Michael Kazakov’s “pstld” is an implementation that worked well for the things I tried (CPU only, but often surprisingly fast)

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)

JÜLICH Forschungszentrum

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against
  - Comparison setup: single-threaded vs. multi-threaded vs. GPU

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against
  - Comparison setup: single-threaded vs. multi-threaded vs. GPU
  - Your actual problem:

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against
  - Comparison setup: single-threaded vs. multi-threaded vs. GPU
  - Your actual problem:
    - Algorithms used (not all are created equal)

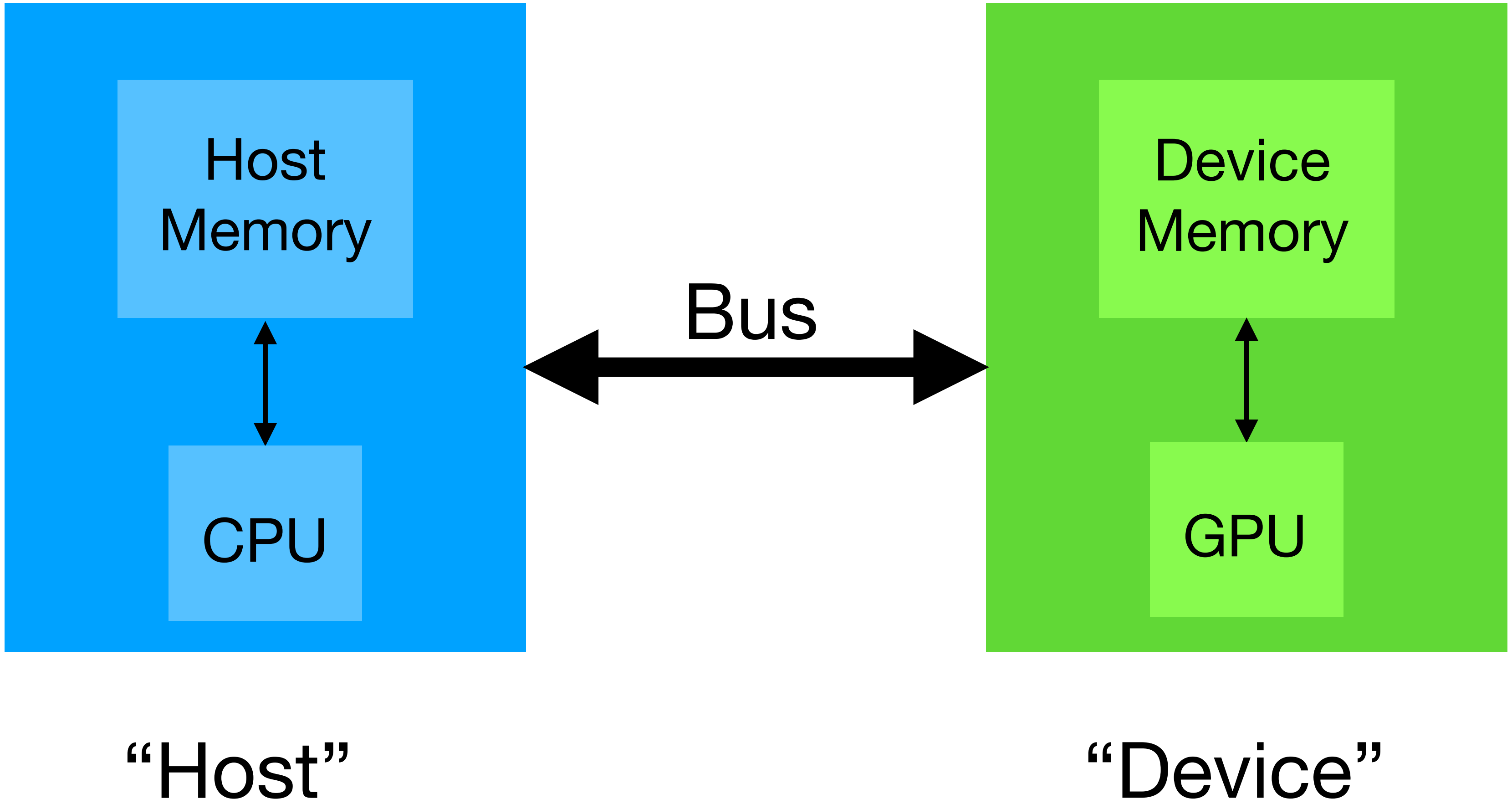
WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against
  - Comparison setup: single-threaded vs. multi-threaded vs. GPU
  - Your actual problem:
    - Algorithms used (not all are created equal)
    - Amount of code in project or region that can be parallelized (data movement)

WHY I WILL NOT SHOW ANY BENCHMARKS IN THIS TALK

- Short answer: Your Mileage May Vary. Significantly.
- Long answer: Speedup depends on too many individual factors:
  - Make and model of GPU used (50 or more x difference in runtime between Desktop and Data center GPUs)
  - CPU system compared against
  - Comparison setup: single-threaded vs. multi-threaded vs. GPU
  - Your actual problem:
    - Algorithms used (not all are created equal)
    - Amount of code in project or region that can be parallelized (data movement)
    - Problem size (efficiency)

GPU BASICS: A TYPICAL GPU SYSTEM AT A GLANCE



ABOUT A TYPICAL GPU SYSTEM

- The computer containing and/or controlling the GPU is usually called the “Host”
- The GPU is usually called the “Device”
- They are connected by a bus
- Host and Device usually have their own, disjoint memory
- Host and Device usually have different (read: incompatible) machine code

WHY ARE GPUS FAST?

Calculation

- GPUs can perform thousands of threads in parallel - with limitations:
  - Threads are grouped in blocks
    - Block size is, to some extend, user-defined
    - Limited synchronization and data exchange within the block (shared memory)
    - Order of execution of blocks is undefined
    - No data exchange and synchronization between blocks
    - A certain number threads always operates in lockstep, doing either the same or nothing
      - 32 for Nvidia (“warp”), 64 for AMD (“wavefront”)
      - Good for synchronization, but possible performance pitfall
      - Direct data exchange between threads of the same warp/wavefront

WHY ARE GPUS FAST?

Memory Subsystems

- Device memory usually is significantly faster than host memory, but
  - Exchange between Host- and Device memory is slow (bus transfer)
    - Slowdown depends on bus architecture
  - Device memory caches are relatively small (compared to CPUs)
  - More threads may concurrently access Device memory
    - Certain access patterns are preferred (coalesced access - neighboring threads read neighboring data)

THE SAMPLE TASK

Fill a vector with discrete values following one period of the sine-function

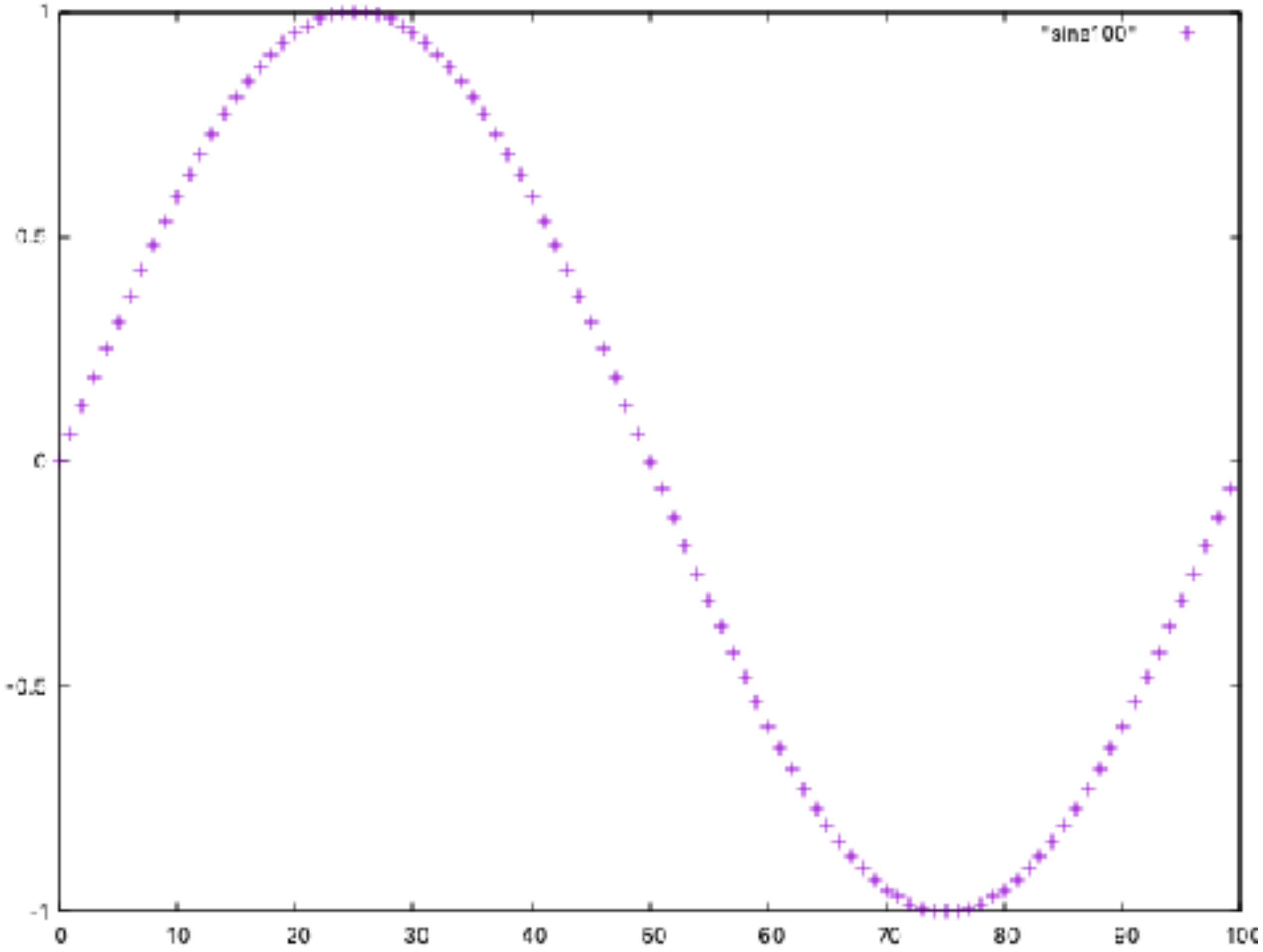
```
template<typename T, typename Allocator>
void calc_wave(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    std::for_each_n(std::views::iota(0).begin(),n,
        [&](int i){
            wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<float> / n);
    });
}
```

Note that, for the sake of simplicity, there is no implementation of or checking for return values (error codes), exceptions etc. in the sample sample code shown in this talk. The CUDA/HIP library functions shown usually return a status code, (parallel) algorithms may throw exceptions.

JÜLICH Forschungszentrum

THE SAMPLE TASK

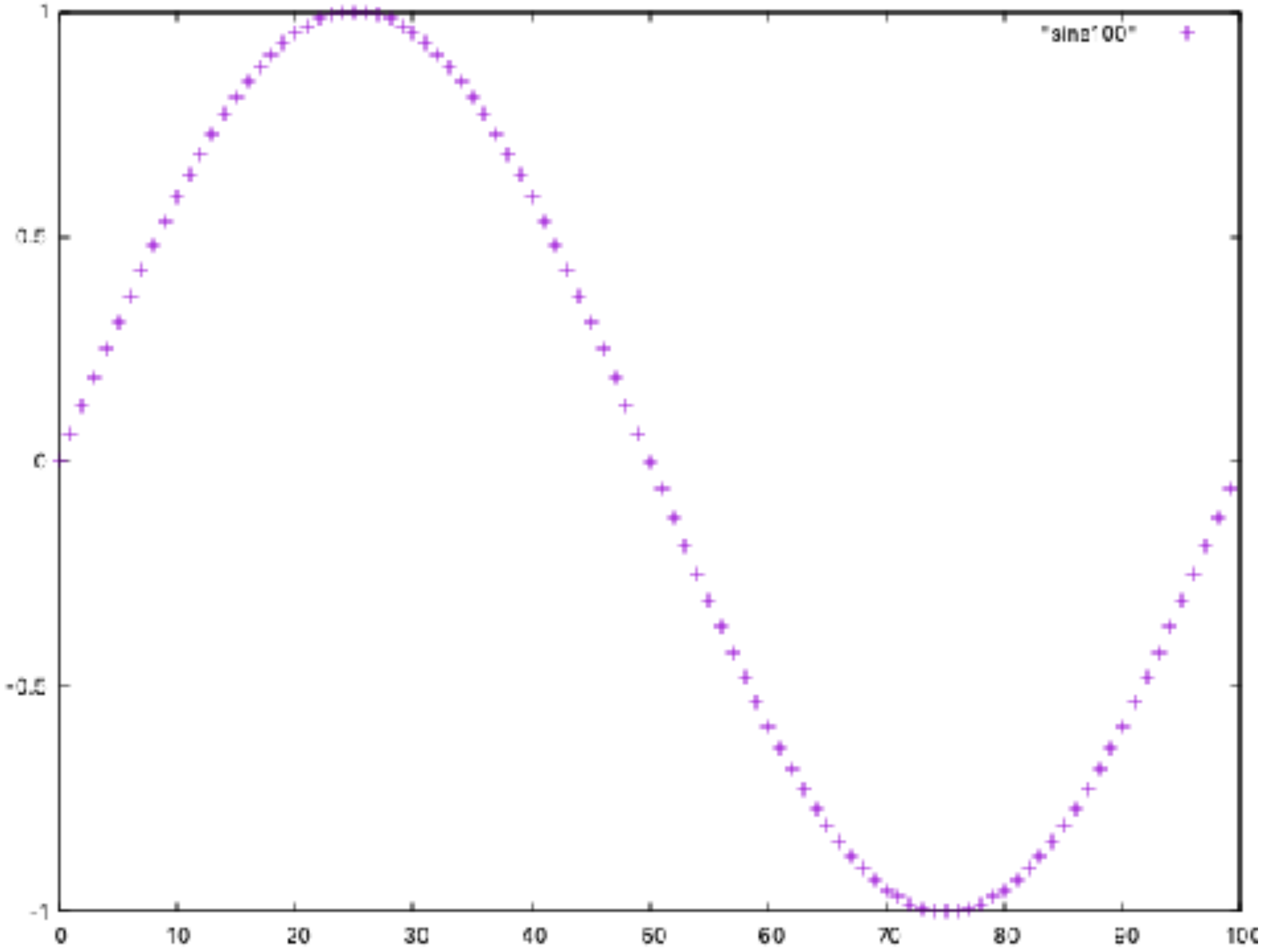
Fill a vector with discrete values following one period of the sine-function



JÜLICH
Forschungszentrum

THE SAMPLE TASK

Fill a vector with discrete values following one period of the sine-function



JÜLICH
Forschungszentrum

“CLASSIC” GPU PROGRAMMING MODEL EXAMPLES

“CLASSIC” GPU PROGRAMMING MODEL EXAMPLES

- Kernel-based / language extensions:
  - CUDA (Nvidia) / HIP (AMD)
  - OpenCL

“CLASSIC” GPU PROGRAMMING MODEL EXAMPLES

- Kernel-based / language extensions:
  - CUDA (Nvidia) / HIP (AMD)
  - OpenCL
- Pragma-based:
  - OpenMP
  - OpenACC

“CLASSIC” GPU PROGRAMMING MODEL EXAMPLES

- Kernel-based / language extensions:
  - CUDA (Nvidia) / HIP (AMD)
  - OpenCL
- Pragma-based:
  - OpenMP
  - OpenACC
- Library-based
  - SYCL
  - Kokkos

“CLASSIC” GPU PROGRAMMING MODEL EXAMPLES

- Kernel-based / language extensions:
  - CUDA (Nvidia) / HIP (AMD)
  - OpenCL
- Pragma-based:
  - OpenMP
  - OpenACC
- Library-based
  - SYCL
  - Kokkos
- …

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)[1]

- Superset of a subset of (currently) C++20
  - Code to be accelerated is written in special functions named “kernels”
  - Kernels are called using a special notation specifying threading and resource configuration
    - requires at least some knowledge about the inner workings of a GPU
- Extensive API for memory and hardware management
- Libraries providing accelerated functions implemented for many standard purposes (linear algebra, FFT…)
- Ecosystems of varying complexity for debugging, benchmarking, profiling etc.

1) HIP is part of AMDs RocM ecosystem. It is syntactically and logically very close to CUDA and also allows compiling code for Nvidia GPUs (not vice versa). Due to technical differences, some code adjustments may be necessary, not just for performance

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

The Language Superset and Library Functions

- Kernels are functions labeled __global__ and always return void
- User-defined functions to be called from kernels are labeled __device__
- Kernels can access special variables related to the current thread ID
- User-defined functions to be called in kernels and from host code are labeled __host__ __device__
- Experimental compiler option allow constexpr functions to be called from kernels
  - allows reuse of code from/in non-CUDA environments, but compilation and/or execution may (sometimes silently!) fail, if non-constexpr code is called
- Memory-, Thread- and Hardware-Management functions, types etc. have, where applicable, the same name prefixed by cuda or hip, i.e. `cudaMalloc` vs. `hipMalloc`

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```


JÜLICH Forschungszentrum

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

JÜLICH Forschungszentrum

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

additional data elements must be allocated on the device

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

JÜLICH Forschungszentrum

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}
```

__global__ functions are code to be executed on the Device ("kernels")

split the problem into a number of independent blocks of equal thread-count

```
auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

thread IDs come as an up to 6D-grid and need to linearized

split the problem into a number of independent blocks of equal thread-count

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

special variables only available in global/device code

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

thread IDs come as an up to 6D-grid and need to linearized

split the problem into a number of independent blocks of equal thread-count

```
auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

special variables only available in global/device code

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}
```

__global__ functions are code to be executed on the Device (“kernels”)

thread IDs come as an up to 6D-grid and need to linearized

split the problem into a number of independent blocks of equal thread-count

```
auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

wait for asynchronous kernel call to complete

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

special variables only available in global/device code

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }
```

__global__ functions are code to be executed on the Device (“kernels”)

thread IDs come as an up to 6D-grid and need to linearized

```
template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}
```

split the problem into a number of independent blocks of equal thread-count

```
auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }
```

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

wait for asynchronous kernel call to complete

results need to be copied from device to host

```
template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

PROGRAMMING MODELS: CUDA (NVIDIA) / HIP (AMD)

GPU programming (almost like) in the days of yore

special variables only available in global/device code

```
__device__ auto calc_linear_thread_id() { return blockIdx.x*blockDim.x+threadIdx.x; }

template<typename T>
__global__ void make_wave_kernel(int n, T *wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}
```

__global__ functions are code to be executed on the Device ("kernels")

thread IDs come as an up to 6D-grid and need to linearized

split the problem into a number of independent blocks of equal thread-count

```
auto n_blocks(int n_threads, unsigned block_size) { return (n_threads+block_size-1)/block_size; }

template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

additional data elements must be allocated on the device

<<< >>>-notation for kernel call and configuration

wait for asynchronous kernel call to complete

results need to be copied from device to host

free device data

PROGRAMMING MODELS: OPENMP

- Pragma-based
- Mostly loop parallelization
- Has been around for a long time for parallel CPU computing
- Additional pragma keywords for performing code on accelerators:
  - Introduction of accelerator region (“target”)
  - Memory management (“map”)
  - Thread management (“teams”, “distribute”, …)
  - …
- Parallel target regions are translated into kernels
- Limited support for C++

PROGRAMMING MODELS: OPENMP

Short, but more C than C++

```
template<typename T, typename Allocator>
void calc_wave_omp(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
#pragma omp target map(from:wave_data[:n])
#pragma omp teams distribute
    for (unsigned i = 0; i < n; ++i)
        wave_data[i] = sin(i * float(2. * M_PI) / n);
}
```

PROGRAMMING MODELS: OPENMP

Short, but more C than C++

```
template<typename T, typename Allocator>
void calc_wave_omp(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
#pragma omp target map(from:wave_data[:n])
#pragma omp teams distribute
    for (unsigned i = 0; i < n; ++i)
        wave_data[i] = sin(i * float(2. * M_PI) / n);
}
```

enter accelerated region and copy results from device

PROGRAMMING MODELS: OPENMP

Short, but more C than C++

```
template<typename T, typename Allocator>
void calc_wave_omp(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
#pragma omp target map(from:wave_data[:n])
#pragma omp teams distribute
    for (unsigned i = 0; i < n; ++i)
        wave_data[i] = sin(i * float(2. * M_PI) / n);
}
```

enter accelerated region and copy results from device

distribute computation over GPU

PROGRAMMING MODELS: FURTHER CODE EXAMPLES

- If you want to see more, David Olsen did a terrific job in

A TRIP DOWN MEMORY LANE

- The seamless integration that can be achieved today is also the result of the evolution of memory access models for GPUs

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

When things were tedious or slow

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

When things were tedious or slow

- The fast, but tedious way: explicit copying of all data between Host- and Device-memory
  - two sets of data pointers for data on Host and Device
  - manual allocation of Device-memory
  - significant runtime overhead esp. for small amounts of data

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

```
template<typename T, typename Allocator>
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    T* device_data=nullptr;
    cudaMalloc(&device_data,n*sizeof(T));
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,device_data);
    cudaDeviceSynchronize();
    cudaMemcpy(wave_data.data(),device_data,n*sizeof(T),cudaMemcpyDefault);
    cudaFree(device_data);
}
```

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

Pinned Host Memory, a Dual Use Mode of Allocation

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

Pinned Host Memory, a Dual Use Mode of Allocation

- Page locked (pinned) allocation of Host-memory allows faster transfers over the bus

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

Pinned Host Memory, a Dual Use Mode of Allocation

- Page locked (pinned) allocation of Host-memory allows faster transfers over the bus
- Amount of pinned memory is limited (overuse degrades system performance)

A TRIP DOWN MEMORY LANE: THE DAYS OF YORE

Pinned Host Memory, a Dual Use Mode of Allocation

- Page locked (pinned) allocation of Host-memory allows faster transfers over the bus
- Amount of pinned memory is limited (overuse degrades system performance)
- ~2009 zero copy access was introduced:
  - direct access by the Device
  - throughput and latency are limited by bus speed, but with less code and API overhead
  - good for i.e. returning scalar results

A TRIP DOWN MEMORY LANE: PINNED MEMORY

Useful, but high latency and slow throughput

```
template<typename T>
struct Pinned_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocHost(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFreeHost(p); }
};

template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>>)
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```


JÜLICH Forschungszentrum

A TRIP DOWN MEMORY LANE: PINNED MEMORY

Useful, but high latency and slow throughput

```
template<typename T>
struct Pinned_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocHost(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFreeHost(p); }
};
```

allocates pinned host memory, accessible (over the bus) from host and device

```
template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>>)
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```

A TRIP DOWN MEMORY LANE: PINNED MEMORY

Useful, but high latency and slow throughput

```
template<typename T>
struct Pinned_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocHost(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFreeHost(p); }
};

template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>>)
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```

allocates pinned host memory, accessible (over the bus) from host and device

no extra data or copy necessary

A TRIP DOWN MEMORY LANE: MANAGED MEMORY

Taking Some of the Burden off the Programmer’s Shoulders

- Introduced ~2013
- Based on Unified Address Space, as opposed to having having separate pointer address spaces for Host and Device
- Software- or (since Pascal architecture, ~2016) hardware-implemented automatic migration of memory pages between Host and Device
  - Hardware implementation allows oversubscription of Device memory
- Managed Memory must be explicitly allocated as such
  - Only for dynamic allocations (heap memory)
  - Can be set as default for the relevant compilers
- Actual location of memory on Host or Device is usually determined by “first touch” (since Pascal)
- No concurrent access from Host and Device

A TRIP DOWN MEMORY LANE: MANAGED MEMORY

Automatically moved where it's needed

```
template<typename T>
struct Managed_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocManaged(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFree(p); }
};

template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>> ||
          std::same_as<Allocator,Managed_allocator<T>> )
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```

A TRIP DOWN MEMORY LANE: MANAGED MEMORY

Automatically moved where it's needed

```
template<typename T>
struct Managed_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocManaged(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFree(p); }
};

template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>> ||
          std::same_as<Allocator,Managed_allocator<T>> )
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```

allocates managed memory that is automatically moved between host and device, usually via paging

JÜLICH
Forschungszentrum

A TRIP DOWN MEMORY LANE: MANAGED MEMORY

Automatically moved where it's needed

```
template<typename T>
struct Managed_allocator {
    using value_type=T;
    T* allocate(size_t n) {
        T *ptr=nullptr;
        cudaMallocManaged(&ptr,n*sizeof(T));
        return ptr;
    }
    void deallocate(T* p, size_t n) { cudaFree(p); }
};
```

allocates managed memory that is automatically moved between host and device, usually via paging

```
template<typename T, typename Allocator>
requires (std::same_as<Allocator,Pinned_allocator<T>> ||
          std::same_as<Allocator,Managed_allocator<T>> )
void calc_wave_cuda(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data.data());
    cudaDeviceSynchronize();
}
```

wave_data might be stack allocated, so it is only safe to pass its heap allocated data

JÜLICH Forschungszentrum

A TRIP DOWN MEMORY LANE: HMM

Heterogeneous Memory Management

- Introduced ~2022
- All system memory can be accessed (again via paging) to/from Host and Device
  - no special allocation necessary
  - works with heap <u>and</u> stack memory
- Hardware and OS kernel/driver support necessary
  - Linux support on fairly recent kernels
  - Windows support using WSL (at least according to google AI, couldn't verify this myself or from the cited sources)
  - works with most Nvidia consumer and data center GPUs introduced ~2019 or later
  - works with fairly recent AMD data center GPUs (Instinct MI100 or later)

A TRIP DOWN MEMORY LANE: HMM

Heterogeneous Memory Management

```
template<typename T>
__global__ void make_wave_kernel(int n, std::vector<T>& wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

template<typename T>
void calc_wave_cuda(std::vector<T>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data);
    cudaDeviceSynchronize();
}
```


JÜLICH
Forschungszentrum

A TRIP DOWN MEMORY LANE: HMM

Heterogeneous Memory Management

```
template<typename T>
__global__ void make_wave_kernel(int n, std::vector<T>& wave_data) {
    auto i=calc_linear_thread_id();
    if (i<n)
        wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<T> / n);
}

template<typename T>
void calc_wave_cuda(std::vector<T>& wave_data) {
    auto n=wave_data.size();
    static constexpr int block_size=512;
    make_wave_kernel<<<n_blocks(n,block_size),block_size>>>(n,wave_data);
    cudaDeviceSynchronize();
}
```

stack and heap allocations are both accessible from host and device so we can just pass the reference

JÜLICH Forschungszentrum

A FINAL WARNING ABOUT MANAGED MEMORY AND HMM

You may not invoke or even see data transfers, but you still have to pay for them!

C++ PARALLEL ALGORITHMS

- Extension to many of C++’s standard algorithms
- Introduced in C++17
- Additional first parameter denoting the execution policy (very simplified)
  - std::execution::seq: sequential execution
  - std::execution::par: parallel execution
  - std::execution::par_unseq: limited functionality parallel execution (no locks etc.)
  - std::execution::unseq: single-threaded parallel, i.e. vectorization
- GPU parallelization uses std::execution::par_unseq
- Example: std::sort(std::execution::par_unseq, v.begin(), v.end())

WHAT DOES PARALLEL_UNSEQUENCED_POLICY MEAN?

Code running within a par_unseq section (non exhaustive)

- must be thread-safe (data races!)
- may perform
  - no memory allocation/deallocation
  - only lock-free atomics
  - no synchronization using mutexes etc.

ADDITIONAL LIMITATIONS FOR GPU CODE

- No access to external devices (I/O)
- No exception throwing or handling
- No streams (limited printf possible)
- No passing of function pointers between Host and Device
  - This includes classes with dynamic polymorphism
- No target control beyond the execution_policy: if you label it par_unseq, the compiler will put it on GPU (or die trying)
- No calls to libraries that are compiled without support for the respective architecture and memory mode
- Iterators passed must be random access iterators

SAMPLE TASK, READY FOR GPU ACCELERATION

For systems using C++20 and HMM

```
template<typename T, typename Allocator>
void calc_wave(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    std::for_each_n(std::views::iota(0).begin(),n,
        [&](int i){
            wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<float> / n);
    });
}
```

SAMPLE TASK, READY FOR GPU ACCELERATION

For systems using C++20 and HMM

```
template<typename T, typename Allocator>
void calc_wave_stdpar(std::vector<T,Allocator>& wave_data) {
    auto n=wave_data.size();
    std::for_each_n(std::execution::par_unseq,std::views::iota(0).begin(),n,
        [&](int i){
            wave_data[i]=sin(i * T(2.) * std::numbers::pi_v<float> / n);
    });
}
```

one single change

SAMPLE TASK, READY FOR GPU ACCELERATION

For systems using C++17 and Managed Memory

```
template<typename T>
void calc_wave_cpp17noHMM(std::vector<T>& wave_data) {
    auto n=wave_data.size();
    auto data=wave_data.data();
    std::for_each(std::execution::par_unseq,wave_data.begin(), wave_data.end(),
        [=](auto &a){
            auto i=&a-data;
            a=std::sin(i*T(2.*M_PI)/n);
    });
}
```


JÜLICH Forschungszentrum

SAMPLE TASK, READY FOR GPU ACCELERATION

For systems using C++17 and Managed Memory

```
template<typename T>
void calc_wave_cpp17noHMM(std::vector<T>& wave_data) {
    auto n=wave_data.size();
    auto data=wave_data.data();
    std::for_each(std::execution::par_unseq,wave_data.begin(), wave_data.end(),
        [=](auto &a){
            auto i=&a-data;
            a=std::sin(i*T(2.*M_PI)/n);
    });
}
```

wave_data might be stack allocated and therefore inaccessible on the device

SAMPLE TASK, READY FOR GPU ACCELERATION

For systems using C++17 and Managed Memory

```
template<typename T>
void calc_wave_cpp17noHMM(std::vector<T>& wave_data) {
    auto n=wave_data.size();
    auto data=wave_data.data();
    std::for_each(std::execution::par_unseq,wave_data.begin(), wave_data.end(),
        [=](auto &a){
            auto i=&a-data;
            a=std::sin(i*T(2.*M_PI)/n);
    });
}
```

wave_data might be stack allocated and therefore inaccessible on the device

calculate index from pointer difference (no std::views::iota in C++17)

SUPPORTED ALGORITHMS

Currently, most C++ standard algorithms are supported **except** the following (subject to change, may differ between Nvidia and AMD)

- find_end, find_first_of, inplace_merge, is_heap, is_heap_until, nth_element, partial_sort, partial_sort_copy, rotate (sorry, Sean!), rotate_copy, search, search_n, shift_left, shift_right

BEYOND THE STANDARD

Nvidia's nvc++

- Support for other programming models (also in the same translation unit)
  - CUDA (kernels and API)
    - Single pass compiler, architecture dependent coding is redesigned (no __CUDA_ARCH__)
    - Allows special CUDA commands (i.e. atomics) in regular code
  - OpenMP
  - OpenACC
- Compiler interception of heap allocations to ease managed memory support (optional)
- Macros __NVCOMPILER_STDPAR_GPU and __NVCOMPILER_STDPAR_MULTICORE
- New "if target" construct allows code specialization for different hardware at function level

BEYOND THE STANDARD

AMD’s amdclang++

- Compiler interposition of heap allocations to ease managed memory support (optional)
- Macro __HIPSTDPAR__ indicating enabled support for standard parallelism
- There is probably more, pardon my ignorance

CORRECTNESS PITFALLS

- Data races
  - concurrent writes
  - modification of containers
    - member functions throwing exceptions (i.e. push_back) will not compile
    - others (i.e. pop_back) will just fail
  - Concurrent access to the same data from host and device
- Loss of exceptions

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```


JÜLICH
Forschungszentrum

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

```
int count(0),atomic_ref_count(0),cuda_atomic_count(0);
std::atomic<int> atomic_count(0);
std::vector<int> v0, vn(n,0);
v0.reserve(n);
std::for_each_n(std::execution::par_unseq,std::views::iota(0u).begin(),n,[&](unsigned){
    ++count; // compiles, but does not work correctly (data race)
    ++std::atomic_ref(atomic_ref_count); // works
    ++atomic_count; // works
#if defined(__NVCOMPILER_STDPAR_GPU)
        if target (nv::target::is_device) {
            atomicAdd(&cuda_atomic_count,1); // works with nvc++ only
    }
#endif
    v0.push_back(0); // fails to compile (push_back may throw)
    vn.pop_back(); // compiles, but does not work correctly (data race)
});
```

DATA RACES AND OTHER LIMITATIONS

Possible Result:

```
count:                expected: 1000000   got:      518
atomic_ref_count:     expected: 1000000   got:  1000000
atomic_count:         expected: 1000000   got:  1000000
cuda_atomic_count:    expected: 1000000   got:  1000000
vn.size():            expected:       0   got:   999631
```

PERFORMANCE PITFALLS

- Alternating use of large amounts of data on Host and Device

```
std::sort(std::execution::par_unseq,v.begin(),v.end());
std::for_each(v.begin(),v.end(),
      [](auto i){ if (i&1) std::cout<<i<<" is really odd!\n"; });
std::cout<<std::reduce(std::execution::par_unseq,v.begin(),v.end());
```

- Repeated construction of containers on the Host for Device use

```
for(int j=0;j<10;++j) {
    std::vector<int> v(n,0);
    std::for_each_n(std::execution::par_unseq,std::views::iota(0).begin(),n,
        [&](int i){v[i]=some_func(i,j);});
    do_something_with(v);
}
```

- Double precision performance may be significantly slower on certain GPUs

JÜLICH Forschungszentrum

PERFORMANCE PITFALLS

- Divergent branching, especially
  - nested
  - expensive branches that are relatively rare, but “well distributed”

```
std::for_each_n(std::execution::par_unseq,std::views::iota(0).begin(),n,
    [&](auto i){ if (i%32==0) do_something_expensive(); });
```

- Excessive use of atomic operations
- Small problem sizes that can not efficiently fill the Device

```
std::sort(std::execution::par_unseq,v.begin(),v.begin()+5);
```

- Concurrent use of data on the same memory page by host and device (rare, but tricky)

BEYOND THE STANDARD

Nvidia’s nvc++

- Example: prefetch vector data to Device (initialized at construction, first touch rule applies)

```
std::vector<float> wave_data(n);
#if defined(__NVCOMPILER_STDPAR_GPU)
    cudaMemPrefetchAsync(wave_data.data(),n*sizeof(float),
        cudaMemLocation(cudaMemLocationTypeDevice,0),0);
#endif
…
```

- Example: use CUDA atomics, if possible (may be faster)

```
#if defined(__NVCOMPILER_STDPAR_GPU)
                if target (nv::target::is_device)
                    atomicAdd(&sum,to_add);
                else
#endif
                    std::atomic_ref(sum)+=to_add;
```

SUPPORTED HARDWARE, OS, LANGUAGE LEVEL ETC.

	Nvidia	AMD
Hardware	Volta architecture (late 2017) and newer	Radeon RX 9060/70, RX/Pro (W)7800/7900, Pro W9700 Instinct MI100+ ([1])
OS	Linux Windows subsystem for Linux (WSL)	Linux WSL
Language Standard	C++17, C++20, C++23 (limited)	C++17 C++20 with ROCm 7?[2]
Memory Model	Managed (all GPUs) HMM(all GPUs, open source driver)	Managed (all GPUs) HMM (Instinct)
Compiler	nvc++ (ex PGI)	amdclang++

(1) list of officially supported hardware

([2]) on my machine (not representative) ROCm 7RC1 compiles C++20, but programs segfault

JÜLICH Forschungszentrum

CONCLUSION

- Accelerated standard algorithms are a potentially powerful tool for developing new accelerated code
- The feasibility of your projects depends on
  - The support of your targeted operating system
  - Language features provided by your hardware vendor
- The effort to port existing code significantly depends on your existing code base
- Besides your hardware, the actually achievable acceleration heavily depends on
  - The actual problem you are trying to solve
  - Your understanding of the potential performance pitfalls
- Writing GPU-accelerated C++-code has never been easier than today. Just try it.

RECOMMENDED VIEWING AND READING

- David Olsen
  - Faster Code Through Parallelism on CPUs and GPUs, CppCon 2019, https://youtu.be/cbbKEAWf1ow
- Bryce Adelstein Lelbach:
  - Inside NVC++ and NVFORTRAN, GTC 2021, https://youtu.be/KhZvrF_w1ak
  - C++ Standard Parallelism, GTC 2023, https://youtu.be/nwrgLH5yAlM
- C++17 parallel algorithms and HIPSTDPAR, https://rocm.blogs.amd.com/software-tools-optimization/hipstdpar/README.html
- nvc++ compiler user's guide, https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html
- Mike Kazakov's pstld library, https://github.com/mikekazakov/pstld

SLIDES AND CODE SAMPLES

- Slides for this talk can be found in the discord channel
- Code examples from this talk: https://godbolt.org/z/xPqTrhWx5

JÜLICH Forschungszentrum

THANK YOU FOR YOUR TIME! QUESTIONS?

?