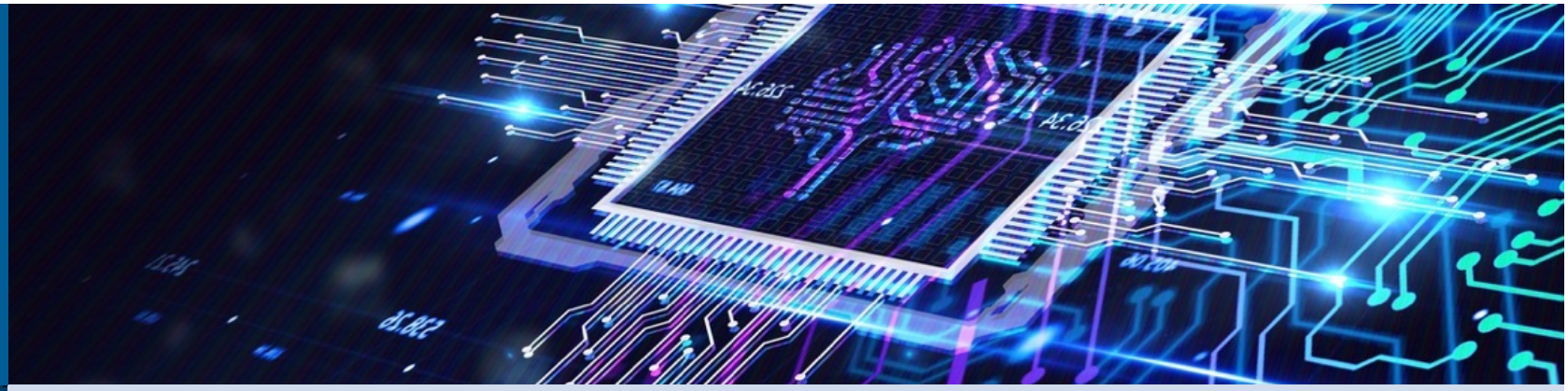
CSC

ICT Solutions for Brilliant Minds



GPU programming with HIP

CSC - IT Center for Science Ltd., Espoo

Cristian Achim, Jaro Hokkanen

November 10-11, 2022

EuroHPC Joint Undertaking

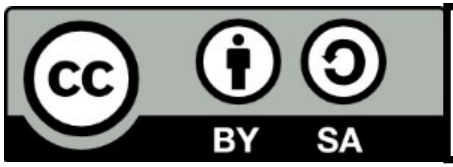
The acquisition and operation of the EuroHPC supercomputer is funded jointly by the EuroHPC Joint Undertaking, through the European Union's Connecting Europe Facility and the Horizon 2020 research and innovation programme, as well as the of Participating States FI, BE, CH, CZ, DK, EE, IS, NO, PL, SE.

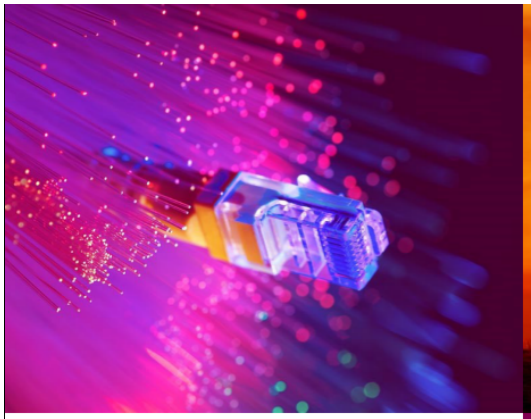
Leverage from the EU 2014–2020

European Union
European Regional Development Fund

REGIONAL COUNCIL OF KAINUU

EURO





Introduction

GPU programming with HIP

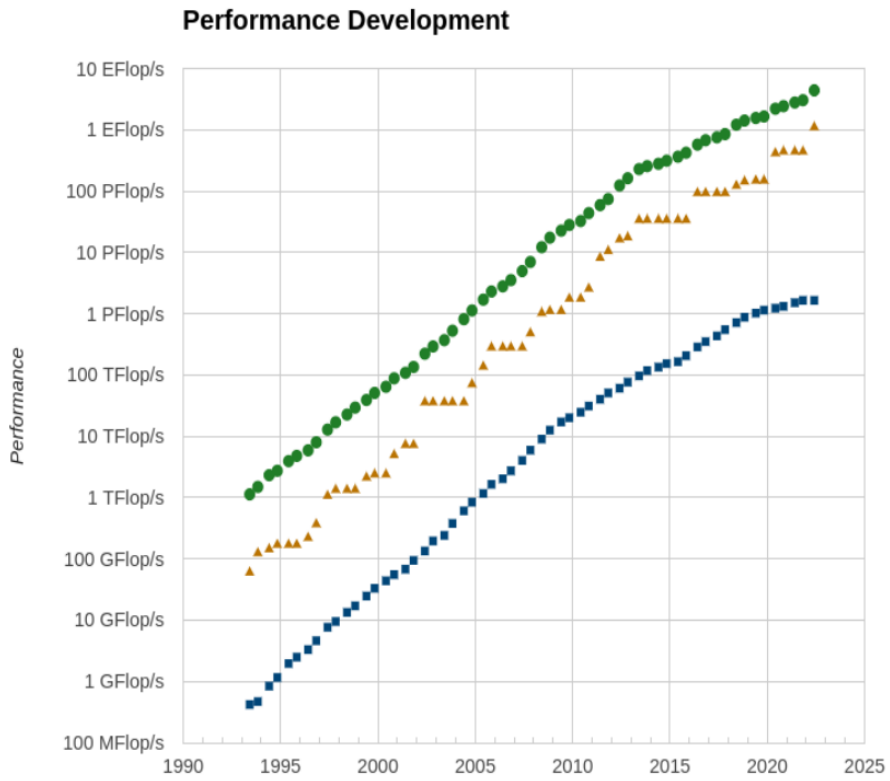
2022-11

CSC Training


CSC – Finnish expertise in ICT for research, education and public administration

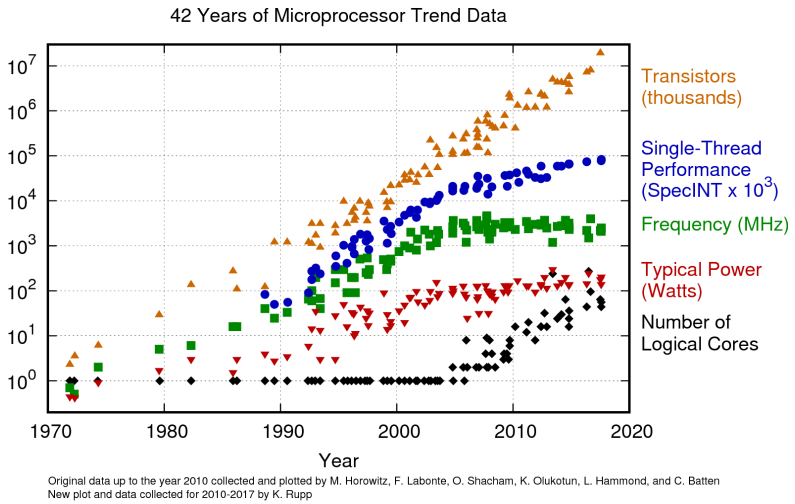
High-performance computing

- High performance computing is fueled by ever increasing performance
- Increasing performance allows breakthroughs in many major challenges that humankind faces today



HPC through the ages

- Achieving performance has been based on various strategies throughout the years
  - Frequency, vectorization, multi-node, multi-core, etc.
- Accelerators provide compute resources based on a very high level of parallelism to reach high performance at low relative power consumption

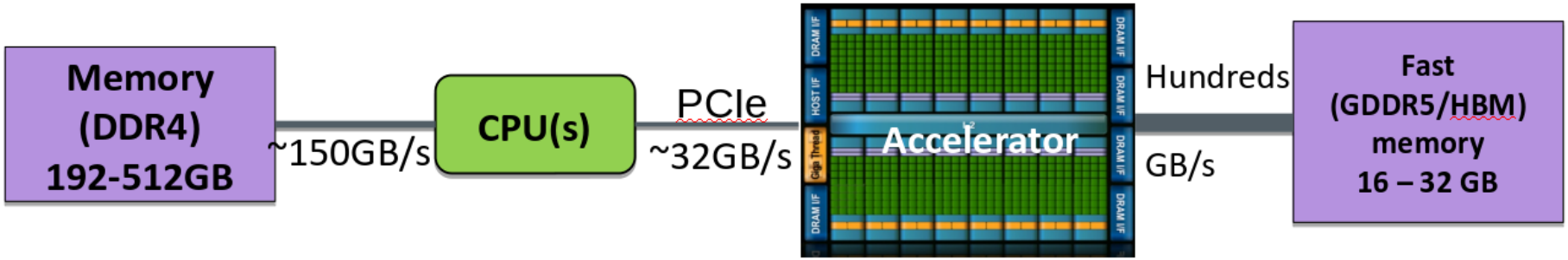


Accelerators

- Specialized parallel hardware for compute-intensive operations
  - Co-processors for traditional CPUs
  - Based on highly parallel architectures
  - Graphics processing units (GPU) have been the most common accelerators during the last few years
- Promises
  - Very high performance per node
- Usually major rewrites of programs required

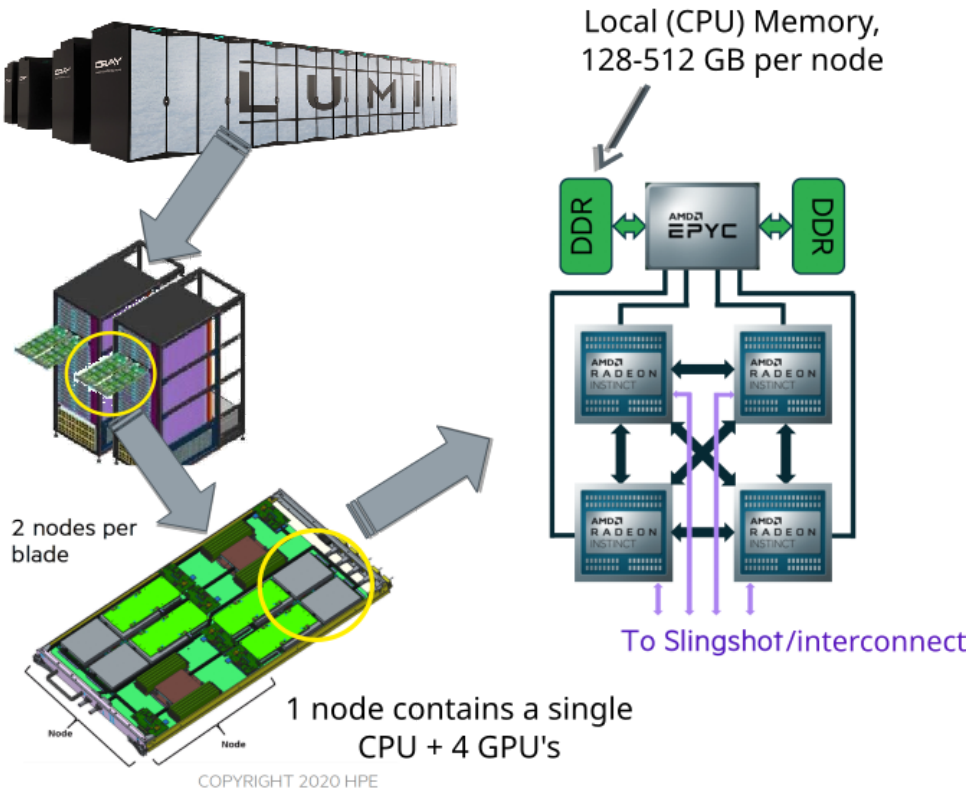
Accelerator model today

- Local memory in GPU
  - Smaller than main memory (32 GB in Puhti, 128 GB in LUMI)
  - Very high bandwidth (up to 3200 GB/s in LUMI)
  - Latency high compared to compute performance



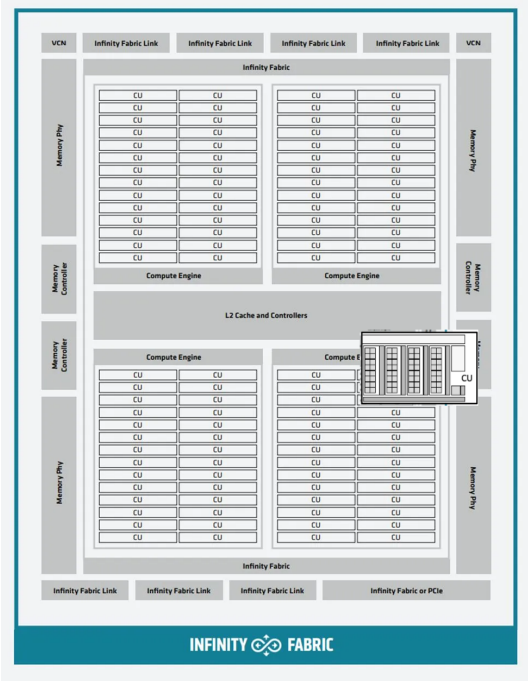
- GPUs are connected to CPUs via PCIe
- Data must be copied from CPU to GPU over the PCIe bus

Lumi - Pre-exascale system in Finland



GPU architecture

- Designed for running thousands or tens of thousands of threads simultaneously
- Running large amounts of threads hides memory access penalties
  - The relative benefit of using a GPU often increases with an increasing job size
- Recurring data movement between CPU and GPU is often a bottleneck
- The penalty for context switching is relatively small



AMD Instinct MI200 architecture (source: AMD).

Challenges in using Accelerators

Applicability: Is your algorithm suitable for GPU?

Programmability: Is the programming effort acceptable?

Portability: Rapidly evolving ecosystem and incompatibilities between vendors.

Availability: Can you access a (large scale) system with GPUs?

Scalability: Can you scale the GPU software efficiently to several nodes?

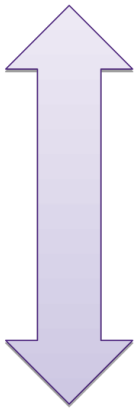
Heterogeneous Programming Model

- GPUs are co-processors to the CPU
- CPU controls the work flow:
  - *offloads* computations to GPU by launching *kernels*
  - allocates and deallocates the memory on GPUs
  - handles the data transfers between CPU and GPUs
- CPU and GPU can work concurrently
  - kernel launches are normally asynchronous

Using GPUs

1. Use existing GPU applications
2. Use accelerated libraries
3. Directive based methods
    - OpenMP, OpenACC
4. Use native GPU language
    - CUDA, **HIP**, SYCL, Kokkos,...

Easier, but more limited



More difficult, but more opportunities

Directive-based accelerator languages

- Annotating code to pinpoint accelerator-offloadable regions
- OpenACC
  - created in 2011, latest version is 3.1 (November 2020)
  - Mostly Nvidia
- OpenMP
  - Earlier only threading for CPUs
  - initial support for accelerators in 4.0 (2013), significant improvements & extensions in 4.5 (2015), 5.0 (2018), 5.1 (2020 and 5.2 (2021)
- Focus on optimizing productivity
- Reasonable performance with quite limited effort, but not guaranteed

Native GPU code: HIP / CUDA

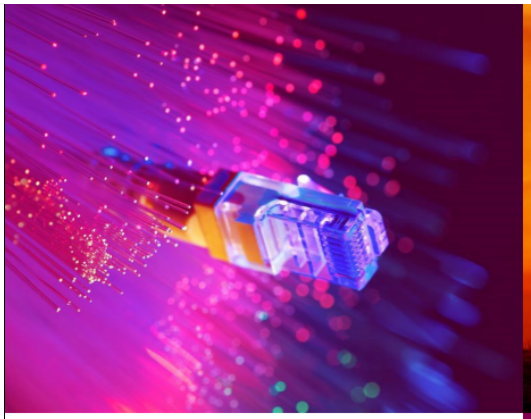
- CUDA
  - has been the *de facto* standard for native GPU code for years
  - extensive set of optimised libraries available
  - custom syntax (extension of C++) supported only by CUDA compilers
  - support only for NVIDIA devices
- HIP
  - AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
  - standard C++ syntax, uses nvcc/hcc compiler in the background
  - almost a one-on-one clone of CUDA from the user perspective
  - ecosystem is new and developing fast

GPUs @ CSC

- **Puhti-AI**: 80 nodes, total peak performance of 2.7 Petaflops
  - Four Nvidia V100 GPUs, two 20-core Intel Xeon processors, 3.2 TB fast local storage, network connectivity of 200Gbps aggregate bandwidth
- **Mahti-AI**: 24 nodes, total peak performance of 2. Petaflops
  - Four Nvidia A100 GPUs, two 64-core AMD Epyc processors, 3.8 TB fast local storage, network connectivity of 200Gbps aggregate bandwidth
- **LUMI-G**: 2560 nodes, total peak performance of 500 Petaflops
  - Four AMD MI250X GPUs, one 64-core AMD Epyc processor, 2x3 TB fast local storage, network connectivity of 800Gbps aggregate bandwidth

Summary

- GPUs can provide significant speed-up for many applications
  - High amount of parallelism required for efficient utilization of GPUs
- GPUs are co-processors to CPUs
  - CPU offloads computations to GPUs and manages memory
- Programming models for GPUs
  - Directive based methods: OpenACC, OpenMP
  - Frameworks: Kokkos, SYCL
  - C++ language extensions: CUDA, HIP



HIP and GPU kernels

GPU programming with HIP

2022-11

CSC Training


CSC – Finnish expertise in ICT for research, education and public administration

HIP

- Heterogeneous-computing Interface for Portability
  - AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
- HIP is a C++ runtime API and kernel programming language
  - standard C++ syntax, uses nvcc/hcc compiler in the background
  - almost a one-on-one clone of CUDA from the user perspective
  - allows one to write portable GPU codes
- AMD offers also a wide set of optimised libraries and tools

HIP kernel language

- Qualifiers: `__device__`, `__global__`, `__shared__`, ...
- Built-in variables: `threadIdx.x`, `blockIdx.y`, ...
- Vector types: `int3`, `float2`, `dim3`, ...
- Math functions: `sqrt`, `powf`, `sinh`, ...
- Arithmetic functions: `atomicAdd`, `atomicMin`, ...
- Intrinsic functions: `__syncthreads`, `__threadfence`, ...

HIP API

- Device init and management
- Memory management
- Execution control
- Synchronisation: device, stream, events
- Error handling, context handling, …

HIP programming model

- GPU accelerator is often called a *device* and CPU a *host*
- Parallel code (kernel) is launched by the host and executed on a device by several threads
- Code is written from the point of view of a single thread
  - each thread has a unique ID

Example: Hello world

```
#include <hip/hip_runtime.h>
#include <stdio.h>

int main(void)
{
    int count, device;

    hipGetDeviceCount(&count);
    hipGetDevice(&device);

    printf("Hello! I'm GPU %d out of %d GPUs in total.\n", device, count);

    return 0;
}
```

AMD GPU terminology

- Compute Unit
  - one of the parallel vector processors in a GPU
- Kernel
  - function launched to the GPU that is executed by multiple parallel workers

AMD GPU terminology (continued)

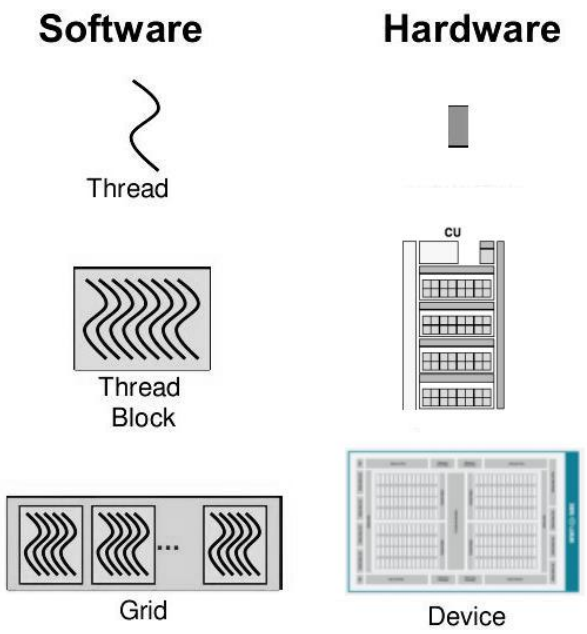
- Thread
  - individual lane in a wavefront
- Wavefront (cf. CUDA warp)
  - collection of threads that execute in lockstep and execute the same instructions
  - each wavefront has fixed number of threads (AMD: 64, NVIDIA 32)
  - number of wavefronts per workgroup is chosen at kernel launch
- Workgroup (cf. CUDA thread block)
  - group of wavefronts (threads) that are on the GPU at the same time and are part of the same compute unit (CU)
  - can synchronise together and communicate through memory in the CU

GPU programming considerations

- GPU model requires many small tasks executing a kernel
  - e.g. can replace iterations of loop with a GPU kernel call
- Need to adapt CPU code to run on the GPU
  - rethink algorithm to fit better into the execution model
  - keep reusing data on the GPU to reach high occupancy of the hardware
  - if necessary, manage data transfers between CPU and GPU memories carefully (can easily become a bottleneck!)

Grid: thread hierarchy

- Kernels are executed on a 3D *grid* of threads
  - threads are partitioned into equal-sized *blocks*
- Code is executed by the threads, the grid is just a way to organise the work
- Dimension of the grid are set at kernel launch



Kernels

- Kernel is a function to be executed by the GPU
- A kernel definition should be of `void` type and must be declared with the `__global__` attribute
- Any function called from a kernel must be declared with `__device__` attribute
- All pointers passed to a kernel should point to memory accessible from the device
- Unique thread and block IDs can be used to distribute work

Example: axpy

```
__global__ void axpy_(int n, double a, double *x, double *y)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < n) {
        y[tid] += a * x[tid];
    }
}
```

- Global ID `tid` calculated based on the thread and block IDs
  - only threads with `tid` smaller than `n` calculate
  - works only if number of threads ≥ `n`

Example: axpy (revisited)

```
__global__ void axpy_(int n, double a, double *x, double *y)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = gridDim.x * blockDim.x;

    for (; tid < n; tid += stride) {
        y[tid] += a * x[tid];
    }
}
```

- Handles any vector size, but grid dimensions should be still “optimised”

Launching kernels

- Kernels are launched with one of the two following options:
  - CUDA syntax (recommended, because it works on CUDA and HIP both):

```
somekernel<<<blocks, threads, shmem, stream>>>(args)
```

  - Native HIP syntax:

```
hipLaunchKernelGGL(somekernel, blocks, threads, shmem, stream, args)
```

- Grid dimensions are obligatory, `shmem`, and `stream` are optional arguments for CUDA syntax, and can be `0` for the native HIP syntax
  - Must have an integer type or vector type of `dim3`
- Kernel execution is asynchronous with the host

Simple memory management

- In order to calculate something on the GPUs, we usually need to allocate device memory and pass a pointer to it when launching a kernel
- Similarly to `cudaMalloc` (or simple `malloc`), HIP provides a function to allocate device memory: `hipMalloc()`

```
double *x_
hipMalloc((void **) &x_, sizeof(double) * n);
```

- To copy data to/from device, one can use `hipMemcpy()`:

```
hipMemcpy(x_, x, sizeof(double) * n, hipMemcpyHostToDevice);
hipMemcpy(x, x_, sizeof(double) * n, hipMemcpyDeviceToHost);
```

Error checking

- Always use HIP error checking with larger codebases!
  - It has low overhead, and can save a lot of debugging time!

```
#define HIP_ERR(err) (hip_error(err, __FILE__, __LINE__))
inline static void hip_error(hipError_t err, const char *file, int line) {
  if (err != hipSuccess) {
    printf("\n\n%s in %s at line %d\n", hipGetErrorString(err), file, line);
    exit(1);
  }
}
```

- Here we wrap a HIP call into the above defined HIP_ERR macro:

```
inline static void* alloc(size_t bytes) {
  void* ptr;
  HIP_ERR(hipMallocManaged(&ptr, bytes));
  return ptr;
}
```

- For simplicity, most exercises of this course do not have error checking, but error checking is strongly recommended when building bigger projects.

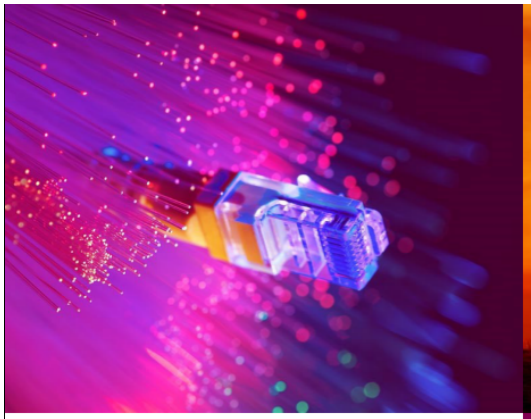
Example: fill (complete device code and launch)

```
#include <hip/hip_runtime.h>
#include <stdio.h>

__global__ void fill_(int n, double *x, double a)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = gridDim.x * blockDim.x;

    for (; tid < n; tid += stride) {
        x[tid] = a;
    }
}
```

```
int main(void)
{
    const int n = 10000;
    double a = 3.4;
    double x[n];
    double *x_;

    // allocate device memory
    hipMalloc(&x_, sizeof(double) * n);

    // launch kernel
    dim3 blocks(32);
    dim3 threads(256);
    hipLaunchKernelGGL(fill_, blocks, threads, 0, 0, n, x_

    // copy data to the host and print
    hipMemcpy(x, x_, sizeof(double) * n, hipMemcpyDeviceTo
    printf("%f %f %f %f ... %f %f\n",
            x[0], x[1], x[2], x[3], x[n-2], x[n-1]);

    return 0;
}
```

Summary

- HIP supports both AMD and NVIDIA GPUs
- HIP contains API functions for writing GPU code
  - Some functions are callable from host, some from device
- Kernels launch grid of blocks, each block consists of many threads
  - Each block is executed in wavefronts of 64 (AMD) or 32 (NVIDIA) threads
- Kernels need to be declared `__global__` and `void` and are launched with special syntaxes



Streams, events, and synchronization

GPU programming with HIP

2022-11

CSC Training

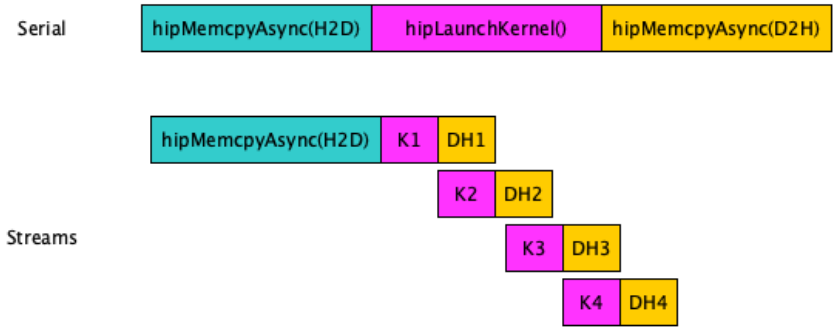
CSC – Finnish expertise in ICT for research, education and public administration

Outline

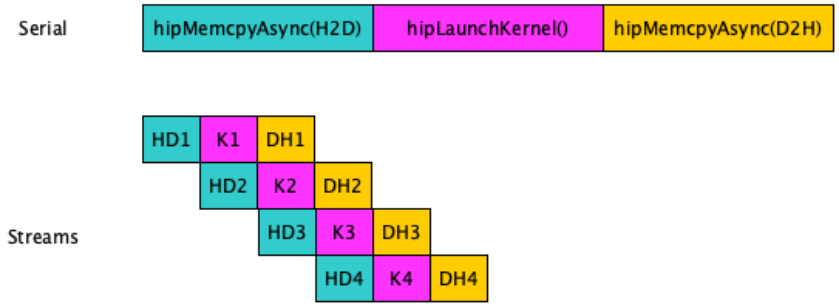
- Streams
- Events
- Synchronization

What is a stream?

- A sequence of operations that execute in order on the GPU
- Operations in different streams may run concurrently if sufficient resources are available



- H-to-D copy runs in a single stream, and the kernel and D-to-H copy are split into 4 streams



- H-to-D copy, kernel, and D-to-H copy are split into 4 streams

Asynchronous funtions and the default stream

- The functions without `Async`-postfix run on the default stream, and are synchronizing with host

```
hipError_t hipMalloc ( void** devPtr, size_t size )
hipError_t hipMemcpy ( void* dst, const void* src, size_t count, hipMemcpyKind kind )
hipError_t hipFree ( void* devPtr )
```

- When using non-default streams, functions with `Async`-postfix are needed
  - These functions take the stream as an additional argument (`0` denotes the default stream)

```
hipError_t hipMallocAsync ( void** devPtr, size_t size, hipStream_t stream )
hipError_t hipMemcpyAsync ( void* dst, const void* src, size_t count, hipMemcpyKind kind, hipStream_t strea
hipError_t hipFreeAsync ( void* devPtr, hipStream_t stream )
```

- Asynchronous memory copies require page-locked host memory (more in Memory lectures)
  - Allocate with `hipMallocHost()` or `hipHostAlloc()` instead of `malloc()`:

```
hipError_t hipMallocHost ( void** ptr, size_t size )
```

```
hipError_t hipHostAlloc ( void** pHost, size_t size, unsigned int  flags )
```

  - Deallocate with `hipFreeHost()`:

```
hipError_t hipFreeHost ( void* ptr )
```

Asynchronisity and kernels

- Kernels are always asynchronous with host, and require explicit synchronization
  - If no stream is specified in the kernel launch, the default stream is used
  - The fourth kernel argument is reserved for the stream
- Running kernels concurrently require placing them in different streams
  - Default stream has special synchronization rules and cannot run concurrently with other streams (applies to all API calls)

```
// Use the default stream
hipkernel<<<grid, block>>>(args);
// Use the default stream
hipkernel<<<grid, block, bytes, 0>>>(args);
// Use the stream strm[i]
hipkernel<<<grid, block, bytes, strm[i]>>>(args);
```

Stream creation, synchronization, and destruction

- Declare a stream variable

```
hipStream_t stream
```

- Create `stream`

```
hipError_t hipStreamCreate ( hipStream_t* stream )
```

- Synchronize `stream`

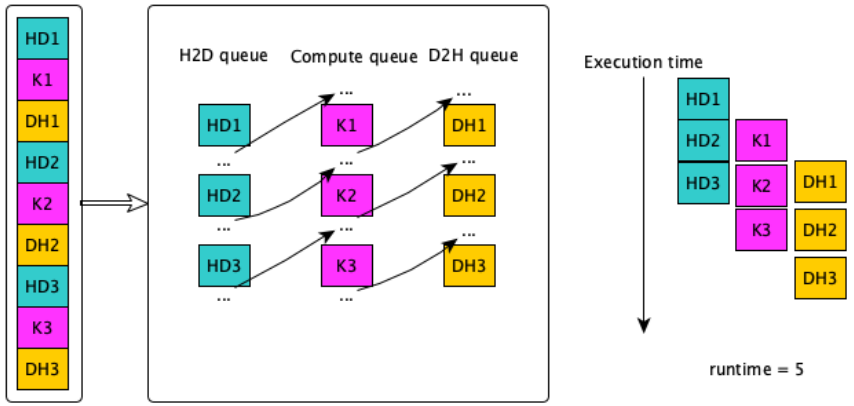
```
hipError_t hipStreamSynchronize ( hipStream_t stream )
```

- Destroy `stream`

```
hipError_t hipStreamDestroy ( hipStream_t stream )
```

Stream example

```
// Declare an array of 3 streams
hipStream_t stream[3];

// Create streams and schedule work
for (int i = 0; i < 3; ++i){
  hipStreamCreate(&stream[i]);

  // Each streams copies data from host to device
  hipMemcpyAsync(d_data[i], h_data[i], bytes,
    hipMemcpyHostToDevice, stream[i]);

  // Each streams runs a kernel
  hipkernel<<<grid, block, 0, strm[i]>>>(d_data[i]);

  // Each streams copies data from device to host
  hipMemcpyAsync(h_data[i], d_data[i],  bytes,
    hipMemcpyDeviceToHost, stream[i]);
}

// Synchronize and destroy streams
for (int i = 0; i < 3; ++i){
  hipStreamSynchronize(stream[i]);
  hipStreamDestroy(stream[i]);
}
```



Events

- Create `event` object

```
hipError_t hipEventCreate ( hipEvent_t* event )
```

- Capture in `event` the contents of `stream` at the time of this call

```
hipError_t hipEventRecord ( hipEvent_t event, hipStream_t stream )
```

- Compute the elapsed time in milliseconds between `start` and `end` events

```
hipError_t hipEventElapsedTime ( float* ms, hipEvent_t start, hipEvent_t end )
```

- Make all future work submitted to `stream` wait for all work captured in `event`

```
hipError_t hipStreamWaitEvent ( hipStream_t stream, hipEvent_t event, unsigned int  flags = 0 )
```

- Wait for `event` to complete

```
hipError_t hipEventSynchronize ( hipEvent_t event )
```

- Destroy `event` object

```
hipError_t hipEventDestroy ( hipEvent_t event )
```

Why events?

- Events provide a mechanism to signal when operations have occurred in a stream
  - Useful for inter-stream synchronization and timing asynchronous events
- Events have a boolean state: occurred / not occurred
  - Important: the default state = occurred

```
// Start timed GPU kernel
clock_t start_kernel_clock = clock();
kernel<<<gridsize, blocksize, 0, stream>>>(d_a, n_total)

// Start timed device-to-host memcopy
clock_t start_d2h_clock = clock();
hipMemcpyAsync(a, d_a, bytes, hipMemcpyDeviceToHost, str

// Stop timing
clock_t stop_clock = clock();
hipStreamSynchronize(stream);
```

- This code snippet can measure how quick the CPU is throwing asynchronous tasks into a queue for the GPU

```
// Start timed GPU kernel
hipEventRecord(start_kernel_event, stream);
kernel<<<gridsize, blocksize, 0, stream>>>(d_a, n_total)

// Start timed device-to-host memcopy
hipEventRecord(start_d2h_event, stream);
hipMemcpyAsync(a, d_a, bytes, hipMemcpyDeviceToHost, str

// Stop timing
hipEventRecord(stop_event, stream);
hipEventSynchronize(stop_event);
```

- This code snippet can measure the duration of each asynchronous task on the GPU

Synchronization

- Synchronize the host with a specific stream

```
hipError_t hipStreamSynchronize ( hipStream_t stream )
```

- Synchronize the host with a specific event

```
hipError_t hipEventSynchronize ( hipEvent_t event )
```

- Synchronize a specific stream with a specific event (the event can be in another stream)

```
hipError_t hipStreamWaitEvent ( hipStream_t stream, hipEvent_t event, unsigned int  flags = 0 )
```

- Synchronize the host with the whole device (wait until all device tasks are finished)

```
hipError_t hipDeviceSynchronize ( void )
```

- In-kernel blockwise synchronization across threads (not between host/device)

```
__syncthreads()
```

Synchronization in a kernel

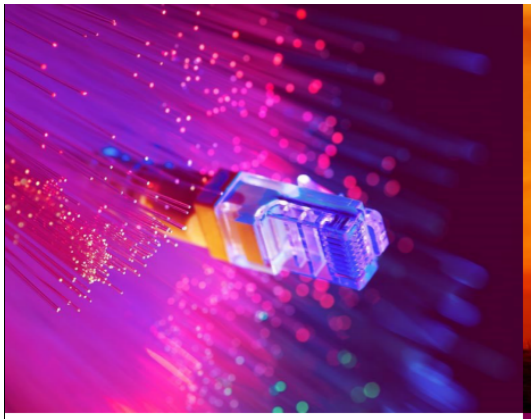
- The device function `__syncthreads()` synchronizes threads within a block inside a kernel
- Often used with shared memory (keyword `__shared__`) which is memory shared between each thread in a block

```
#define BLOCKSIZE 256
__global__ void reverse(double *d_a) {
    __shared__ double s_a[BLOCKSIZE]; /* array of doubles, shared in this block */
    int tid = threadIdx.x;
    s_a[tid] = d_a[tid];              /* each thread fills one entry */
    __syncthreads();                  /* all threads in a block must reach this point before
                                         any thread in that block is allowed to continue. */
    d_a[tid] = s_a[BLOCKSIZE-tid];    /* safe to write out array in reverse order */
}
```

- A simple kernel example for reversing the order of the entries of a block-sized array

Summary

- Streams provide a mechanism to evaluate tasks on the GPU concurrently and asynchronously with the host
  - Asynchronous functions requiring a stream argument are required
  - Kernels are always asynchronous with the host
  - Default stream is by `0` (no stream creation required)
- Events provide a mechanism to signal when operations have occurred in a stream
  - Good for inter-stream sychronization and timing events
- Many host/device synchronizations functions for different purposes
  - The device function `__syncthreads()` is only for in-kernel synchronization between threads in a same block (does not synch threads across blocks)



Memory allocations, access, and unified memory

GPU programming with HIP

2022-11

CSC Training


Outline

- Memory model and hierarchy
- Memory management strategies
- Page-locked memory
- The stream-ordered memory allocator and memory pools

Memory model

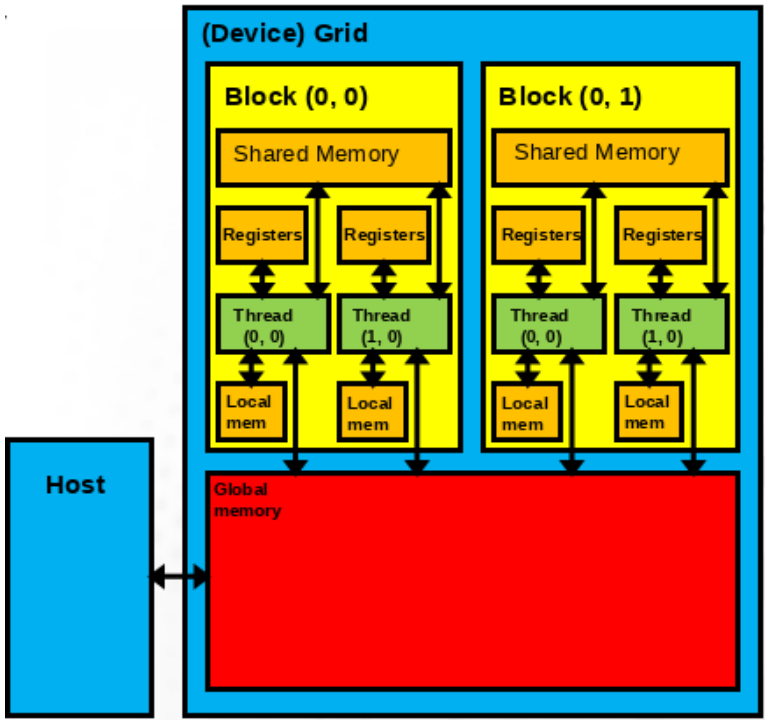
- Host and device have separate physical memories
- It is generally not possible to call malloc() to allocate memory and access the data from the GPU
- Memory management can be
  - Explicit (user manages the movement of the data and makes sure CPU and GPU pointers are not mixed)
  - Automatic, using Unified Memory (data movement is managed in the background by the Unified Memory driver)

Avoid moving data between CPU and GPU

- Data copies between host and device are relatively slow
- To achieve best performance, the host-device data traffic should be minimized regardless of the chosen memory management strategy
  - Initializing arrays on the GPU
  - Rather than just solving a linear equation on a GPU, also setting it up on the device
- Not copying data back and forth between CPU and GPU every step or iteration can have a large performance impact!

Device memory hierarchy

- Registers (per-thread-access)
- Local memory (per-thread-access)
- Shared memory (per-block-access)
- Global memory (global access)



Device memory hierarchy

- Registers (per-thread-access)
  - Used automatically
  - Size on the order of kilobytes
  - Very fast access
- Local memory (per-thread-access)
  - Used automatically if all registers are reserved
  - Local memory resides in global memory
  - Very slow access
- Shared memory (per-block-access)
  - Usage must be explicitly programmed
  - Size on the order of kilobytes
  - Fast access
- Global memory (per-device-access)
  - Managed by the host through HIP API
  - Size on the order of gigabytes
  - Very slow access

Device memory hierarchy (advanced)

- There are more details in the memory hierarchy, some of which are architecture-dependent, eg,
  - Texture memory
  - Constant memory
- Complicates implementation
- Should be considered only when a very high level of optimization is desirable

Important memory operations

Allocate pinned device memory

```
hipError_t hipMalloc(void **devPtr, size_t size)
```

Allocate Unified Memory; the data is moved automatically between host/device

```
hipError_t hipMallocManaged(void **devPtr, size_t size)
```

Deallocate pinned device memory and Unified Memory

```
hipError_t hipFree(void *devPtr)
```

Copy data (host-host, host-device, device-host, device-device)

```
hipError_t hipMemcpy(void *dst, const void *src, size_t count, enum hipMemcpyKind ki
```

Memory management strategies

- Example of explicit memory management

```
int main() {
 int *A, *d_A;
 A = (int *) malloc(N*sizeof(int));
 hipMalloc((void**)&d_A, N*sizeof(int));
 ...
 /* Copy data to GPU and launch kernel */
 hipMemcpy(d_A, A, N*sizeof(int), hipMemcpyHostToDevice);
 kernel<<<...>>>(d_A);
 hipMemcpy(A, d_A, N*sizeof(int), hipMemcpyDeviceToHost);
 hipFree(d_A);
 ...
 printf("A[0]: %d\n", A[0]);
 free(A);
 return 0;
}
```

- Example of Unified Memory

```
int main() {
 int *A;
 hipMallocManaged((void**)&A, N*sizeof(int));
 ...
 /* Launch GPU kernel */
 kernel<<<...>>>(A);
 hipStreamSynchronize(0);
 ...
 printf("A[0]: %d\n", A[0]);
 hipFree(A);
 return 0;
}
```

Unified Memory pros

- Allows incremental development
- Can increase developer productivity significantly
  - Especially large codebases with complex data structures
- Supported by the latest NVIDIA + AMD architectures
- Allows oversubscribing GPU memory on some architectures

Unified Memory cons

- Data transfers between host and device are initially slower, but can be optimized once the code works
  - Through prefetches
  - Through hints
- Must still obey concurrency & coherency rules, not foolproof
- Although the performance on AMD cards is quite good, there may be issues with prefetching and hints (with AMD)

Unified Memory workflow for GPU offloading

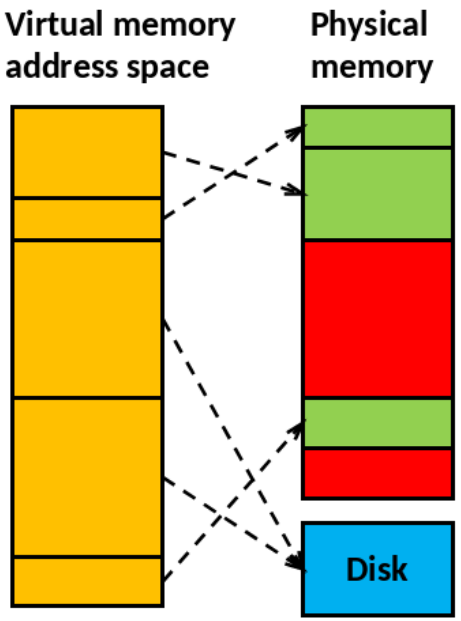
1. Allocate memory for the arrays accessed by the GPU with `hipMallocManaged()` instead of `malloc()`
   - It is a good idea to have a wrapper function and use conditional compilation for memory allocations
2. Offload compute kernels to GPUs
3. Check profiler backtrace for GPU->CPU Unified Memory page-faults (NVIDIA Visual Profiler, Nsight Systems, AMD profiler?)
   - This indicates where the data residing on the GPU is accessed by the CPU (very useful for large codebases, especially if the developer is new to the code)

Unified Memory workflow for GPU offloading

4. Move operations from CPU to GPU if possible, or use hints / prefetching (`hipMemAdvice()` / `hipMemPrefetchAsync()`)
    - It is not necessary to eliminate all page faults, but eliminating the most frequently occurring ones can provide significant performance improvements
5. Allocating GPU memory can have a much higher overhead than allocating standard host memory
    - If GPU memory is allocated and deallocated in a loop, consider using a GPU memory pool allocator for better performance (eg Umpire)

Virtual Memory addressing

- Modern operating systems utilize virtual memory
  - Memory is organized to memory pages
  - Memory pages can reside on swap area on the disk (or on the GPU with Unified Memory)



Page-locked (or pinned) memory

- Normal `malloc()` allows swapping and page faults
- User can page-lock an allocated memory block to a particular physical memory location
- Enables Direct Memory Access (DMA)
- Higher transfer speeds between host and device
- Copying can be interleaved with kernel execution
- Page-locking too much memory can degrade system performance due to paging problems

Allocating page-locked memory on host

- Allocated with `hipMallocHost()` or `hipHostAlloc()` functions instead of `malloc()`
- The allocation can be mapped to the device address space for device access (slow)
  - On some architectures, the host pointer to device-mapped allocation can be directly used in device code (ie, it works similarly to Unified Memory pointer, but the access from the device is slow)
- Deallocated using `hipFreeHost()`

Asynchronous memcopies

- Normal `hipMemcpy()` calls are blocking (ie, synchronizing)
  - The execution of host code is blocked until copying is finished
- To overlap copying and program execution, asynchronous functions are required
  - Such functions have Async suffix, eg, `hipMemcpyAsync()`
- User has to synchronize the program execution
- Asynchronous memory copies require page-locked memory

The stream-ordered memory allocator and memory pools

- Obtain unused memory already allocated from the device's current memory pool in the specified stream (if not enough memory is available, more memory is allocated for the pool)

```
hipError_t hipMallocAsync ( void** devPtr, size_t size, hipStream_t hStream )
```

- Return memory to the pool in the specific stream (does not deallocate memory)

```
hipError_t hipFreeAsync ( void* devPtr, hipStream_t hStream )
```

- By default, the pool is deallocated completely when the stream is synchronized
- The `hipMemPoolAttrReleaseThreshold` represents the size down to which the pool is deallocated during a synchronization, and can be set by

```
hipMemPool_t mempool;
hipDeviceGetDefaultMemPool(&mempool, device);
uint64_t threshold = UINT64_MAX;
hipMemPoolSetAttribute(mempool, hipMemPoolAttrReleaseThreshold, &threshold);
```

- Setting threshold to `UINT64_MAX` means, that the pool is practically never deallocated due to synchronization (because the threshold is a huge number)

Memory pools - Example

- Example 1 - slow

```
for (int i = 0; i < 100; i++) {
  // Allocate memory here (slow)
  hipMalloc(&ptr, size);
  // Run GPU kernel
  kernel<<<..., stream>>>(ptr);
  // Synchronize stream, does not influence memory allocat
  hipStreamSynchronize(stream);
  // Deallocate memory here
  hipFree(ptr);
}
```

- Allocating and deallocating memory in a loop is slow, and can have a significant impact on the performance

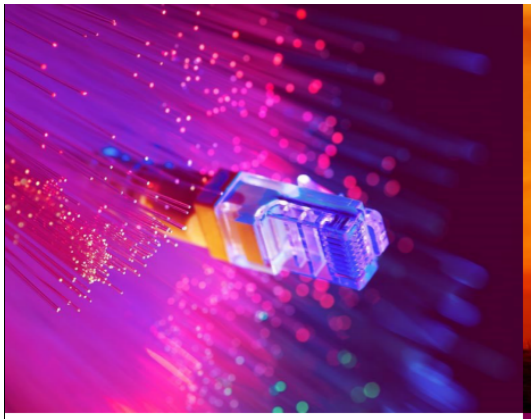
- Example 2 - fast

```
for (int i = 0; i < 100; i++) {
  // Obtain unused memory from the current memory pool,
  // more memory is allocated for the pool if needed
  hipMallocAsync(&ptr, size, stream);
  // Run GPU kernel
  kernel<<<..., stream>>>(ptr);
  // Return memory to the current memory pool
  hipFreeAsync(ptr, stream);
}
// Synchronize and deallocate all memory from the
// current memory pool (default behavior)
hipStreamSynchronize(stream);
```

- Recurring memory allocation and deallocation does not occur anymore, because the memory is obtained from the memory pool and only deallocated during the synchronization (default behavior)

Summary

- Host and device have separate physical memories
- Memory management can be explicit (managed by the user) or automatic (managed by the Unified Memory driver)
- Using Unified Memory can improve developer productivity and result in a cleaner implementation
- The number of data copies between CPU and GPU should be minimized
  - With Unified Memory, if data transfer cannot be avoided, using hints or prefetching to mitigate page faults is beneficial
- Recurring allocation and deallocation is slow, use memory pools instead
  - Libraries provide pooled Unified Memory support as well (eg, Umpire)



Fortran and HIP

GPU programming with HIP

2022-11

CSC Training


CSC – Finnish expertise in ICT for research, education and public administration

Fortran

- No native GPU support in Fortran:
  - HIP functions are callable from C, using `extern C`, compile hipcc
  - interoperability with Fortran via `iso_c_binding`
- Fortran + HIP:
  - link with Fortran
  - needs wrappers and interfaces for all HIP calls
- Hipfort:
  - link with hipcc
  - Fortran Interface For GPU Kernel Libraries
    - HIP: HIP runtime, hipBLAS, hipSPARSE, hipFFT, hipRAND, hipSOLVER
    - ROCm: rocBLAS, rocSPARSE, rocFFT, rocRAND, rocSOLVER
    - memory management: `hipMalloc`, `hipMemcpy`

HIPFort for SAXPY (Y=Y+a*X). Fortran Code

```
program saxpy
  use iso_c_binding
  use hipfort
  use hipfort_check

  implicit none
  interface
     subroutine launch(y,x,b,N) bind(c)
       use iso_c_binding
       implicit none
       type(c_ptr) :: y,x
       integer, value :: N
       real, value :: b
     end subroutine
  end interface

  type(c_ptr) :: dx = c_null_ptr
  type(c_ptr) :: dy = c_null_ptr
  integer, parameter :: N = 400000000
  integer, parameter :: bytes_per_element = 4
  integer(c_size_t), parameter :: Nbytes = N*bytes_pe
  real, allocatable,target,dimension(:) :: x, y
  real, parameter ::  a=2.0
  real :: x_d(N), y_d(N)
```

```
  call hipCheck(hipMalloc(dx,Nbytes))
  call hipCheck(hipMalloc(dy,Nbytes))

  allocate(x(N));allocate(y(N))

  x = 1.0;y = 2.0

  call hipCheck(hipMemcpy(dx, c_loc(x), Nbytes, hipMemcpyHostTo
  call hipCheck(hipMemcpy(dy, c_loc(y), Nbytes, hipMemcpyHostTo

  call launch(dy, dx, a, N)

  call hipCheck(hipDeviceSynchronize())

  call hipCheck(hipMemcpy(c_loc(y), dy, Nbytes, hipMemcpyDevice

  write(*,*) 'Max error: ', maxval(abs(y-4.0))

  call hipCheck(hipFree(dx));call hipCheck(hipFree(dy))

  deallocate(x);deallocate(y)

end program testSaxpy
```

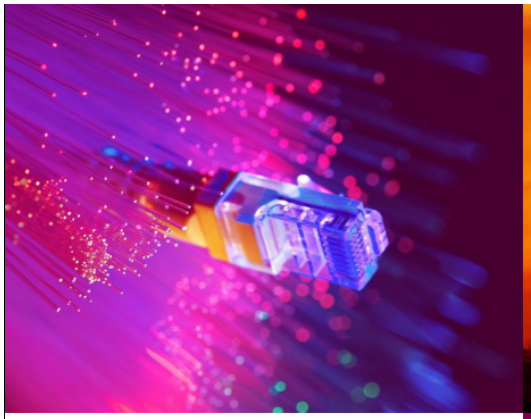
HIPFort for SAXPY (Y=Y+a*X). HIP code

```
#include <hip/hip_runtime.h>
#include <cstdio>

__global__ void saxpy(float *y, float *x,
                      float a, int n)
{
    int i = blockDim.x*blockIdx.x+threadI
    if (i < n) {
      y[i] = y[i] + a*x[i];
    }
}
```

```
extern "C"{
void launch(float **dout, float **da,
            float db, int N)
  {
     dim3 tBlock(256,1,1);
     dim3 grid(ceil((float)N/tBlock.x),1,

    hipLaunchKernelGGL(saxpy,
                       grid, tBlock,
                       0, 0,
                       *dout, *da, db, N)
  }
}
```

Summary

- No native GPU support in Fortran
- HIP functions are callable from C, using `extern C`
  - `iso_c_binding`
  - GPU objects are of type `c_ptr` in Fortran
- Hipfort provides Fortran interfaces for GPU libraries



Kernel optimisation

GPU programming with HIP

2022-11

CSC Training


CSC – Finnish expertise in ICT for research, education and public administration

Libraries (I)

NVIDIA	HIP	ROCm	Description
cuBLAS	hipBLAS	rocBLAS	Basic Linear Algebra Subroutines
cuFFT	hipFFT	rocfft	Fast fourier Transform Library
cuSPARSE	hipSPARSE	rocSPARSE	Sparse BLAS + SMV
cuSOLVER	hipSOLVER	rocSOLVER	Lapack library
AMG-X		rocALUTION	Sparse iterative solvers and preconditioners with Geometric and Algebraic MultiGrid
Thrust	hipThrust	rocThrust	C++ parallel algorithms library

Libraries (II)

NVIDIA	HIP	ROCm	Description
CUB	hipCUB	rocPRIM	Low level Optimized Parallel Primitives
cuDNN		MIOpen	Deep learning solver library
cuRAND	hipRAND	rocRAND	Random number generator library
EIGEN	EIGEN	EIGEN	C++ template library for linear algebra: matrices, vectors, numerical solvers
NCCL		RCCL	Communications Primitives Library based on the MPI equivalents

Kernels

You can call a kernel with the command:

```
hipLaunchKernelGGL(kernel_name, dim3(Blocks), dim3(Threads), 0, 0, arg1, arg2, ...);
```

or

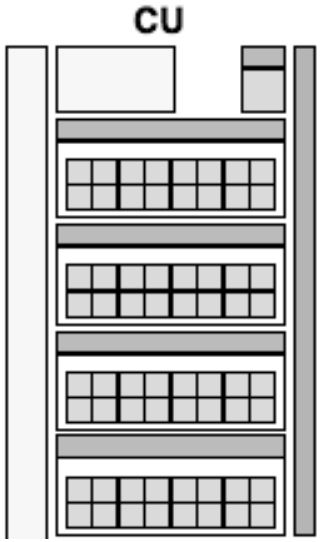
```
kernel_name<<<dim3(Blocks), dim3(Threads),0,0>>>(arg1,arg2,...);
```

- blocks are for the 3D dimensions of the grid of blocks dimensions
- threads for the 3D dimentions of a block of threads
- 0 for bytes of dynamic LDS space, 0 for stream, kernel arguments

Knowledge of the hardware is required for best performance!!!

Compute Units (CU)

- Each AMD CU is a 64-wide execution unit, so multiple of 64 as the thread limit.
  - The 64-wide execution is sub-divided into 4 SIMD units.
  - Each SIMD unit executes a full wavefront instruction in 4 cycles.
  - Heavily dependent of the architecture.



- Minimum 256 threads per block is required for the best performance, in general more tuning (architecture dependent) is required.

Global memory access in device code

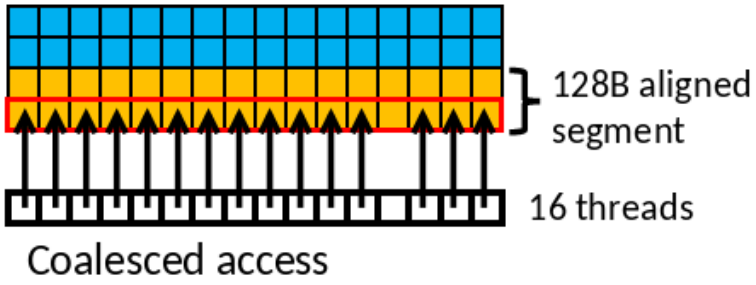
- Global memory access from the device has high latency
- Threads are executed in wavefronts/warps, memory operations are grouped in a similar fashion
- Memory access is optimized for coalesced access where threads read from and write to successive memory locations
- Exact alignment rules and performance issues depend on the architecture

Coalesced memory access

- The global memory loads and stores consist of transactions of a certain size
- If the threads within a wavefront access data within such a block, only one global memory transaction is needed
- Irregular access patterns result in more transactions!

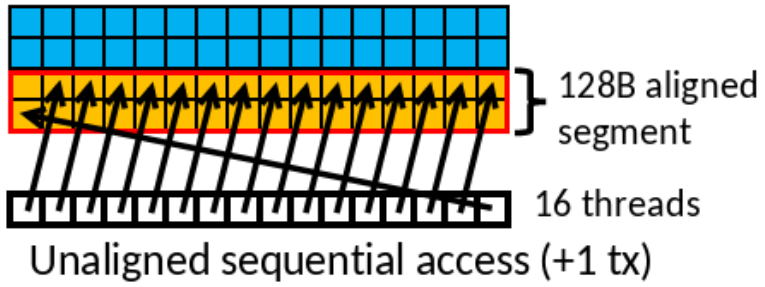
Coalesced memory access example

```
__global__ void memAccess(float *out, flo
{
 int tid = blockIdx.x*blockDim.x + thread
 if(tid != 12) out[tid + 16] = in[tid + 1
}
```



Coalesced access

```
__global__ void memAccess(float *out, flo
{
 int tid = blockIdx.x*blockDim.x + thread
 out[tid + 1] = in[tid + 1];
}
```

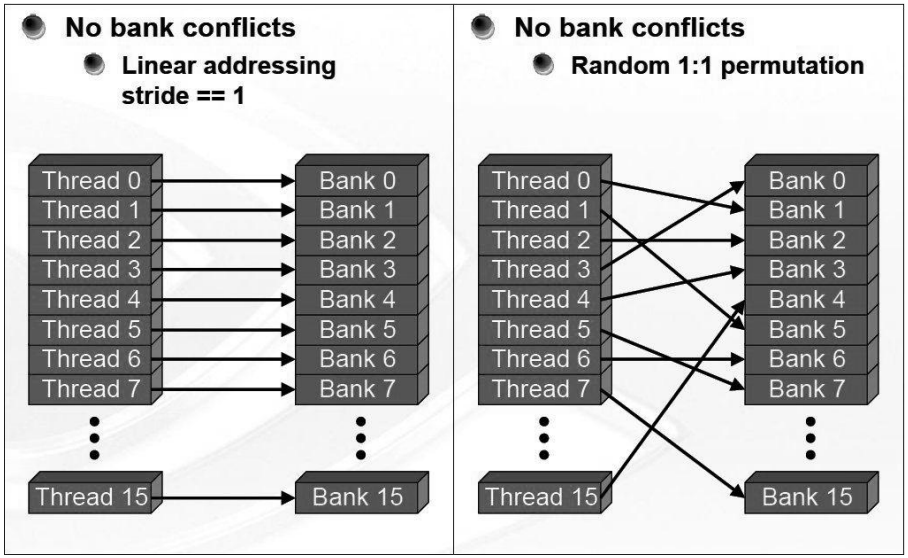


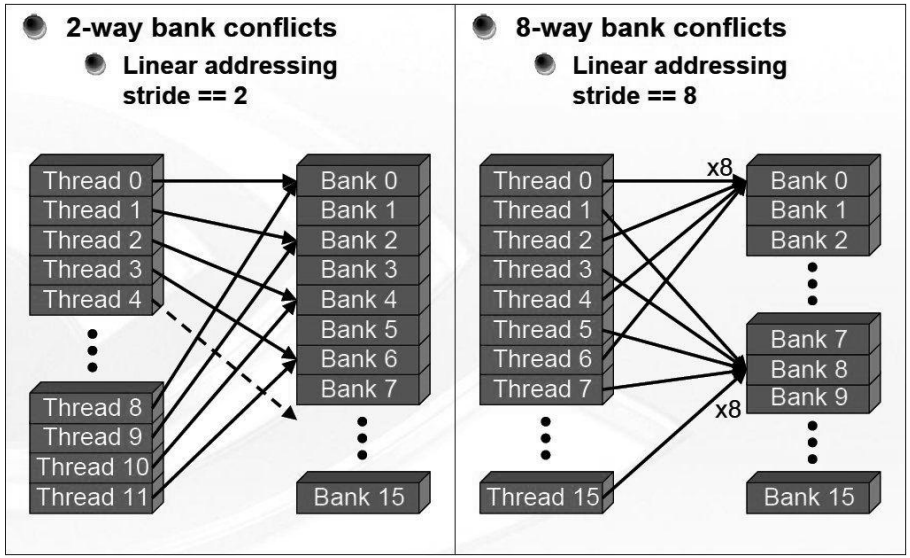
Unaligned sequential access (+1 tx)

Shared memory I

- Fast memory on the CU
- Shared memory is divided into banks
- Each bank can service one address per cycle
- Conflicting accesses are serialized
- Conflicts solved by padding

Shared memory II





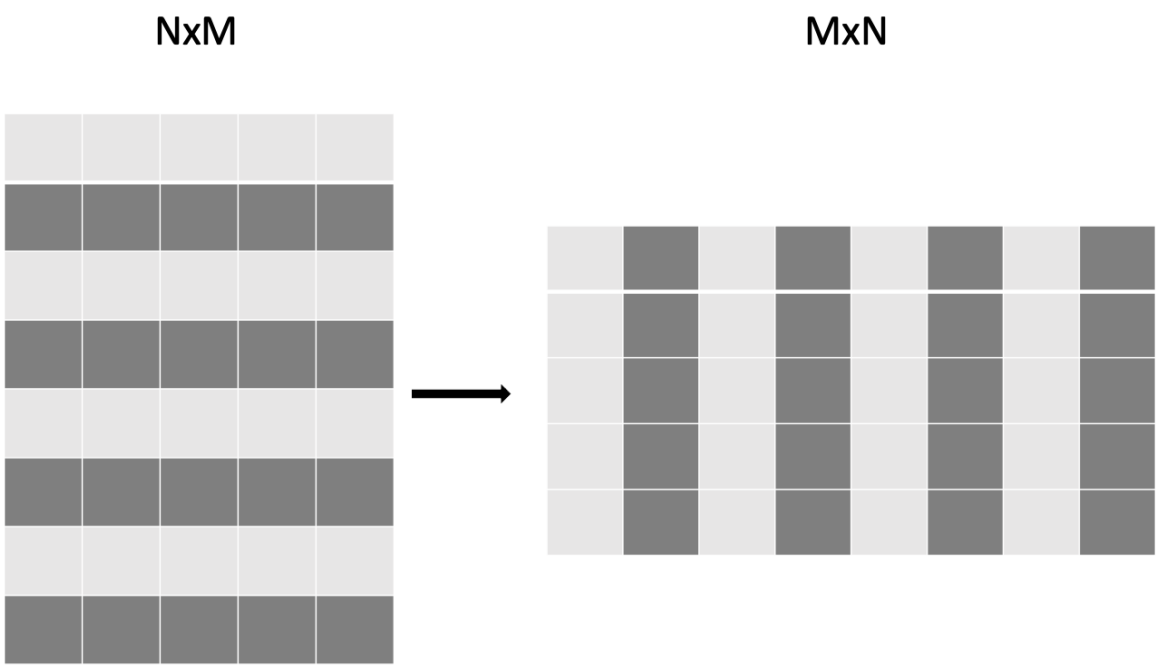
Shared memory III

- Can be used as user controled cache
- Useful to reduce the amount of global memory operations
- Can be used as buffer to transform uncoalesced operations in coalesced
- Limited resources per CU

Low level optimizations

- Avoid branching
  - All threads in wavefront should execute the same instruction
    - `if(tid%2==0)` would result in 2 branches
    - better use `if(tid<N/2)`
- Sometimes recomputing can be faster than reading from the memory
- Depending on the problem, consider using lower precision instead of `double` (math functions are available for `single` and `half` precision)

Optimizing matrix operations. `B(i,j)=A(j,i)`



Copy operation as base

```
__global__ void copy_kernel(float *in, float *out, int width, int height) {
  int x_index = blockIdx.x * tile_dim + threadIdx.x;
  int y_index = blockIdx.y * tile_dim + threadIdx.y;

  int index = y_index * width + x_index;

  out[index] = in[index];
}
```

```
  int block_x = width / tile_dim;
  int block_y = height / tile_dim;
   hipLaunchKernelGGL(copy_kernel, dim3(block_x, block_y),
                      dim3(tile_dim, tile_dim), 0, 0, d_in, d_out, width,
                      height);
   hipDeviceSynchronize();
```

The duration is `0.174 ms` and the effective bandwidth `717 GB/s`

Matrix transpose naive

```
__global__ void transpose_kernel(float *in, float *out, int width, int height) {
  int x_index = blockIdx.x * tile_dim + threadIdx.x;
  int y_index = blockIdx.y * tile_dim + threadIdx.y;

  int in_index = y_index * width + x_index;
  int out_index = x_index * height + y_index;

  out[out_index] = in[in_index];
}
```

The duration is `0.401 ms` and the effective bandwidth `311 GB/s`

Matrix transpose with shared memory

```
__global__ void transpose_lds_kernel(float *in, float *out, int width,
                                     int height) {
  __shared__ float tile[tile_dim][tile_dim];

  int x_tile_index = blockIdx.x * tile_dim;
  int y_tile_index = blockIdx.y * tile_dim;

  int in_index =
      (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.x);
  int out_index =
      (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.x);

  tile[threadIdx.y][threadIdx.x] = in[in_index];

  __syncthreads();

  out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```

The duration is `0.185 ms` and the effective bandwidth `674 GB/s`

Matrix transpose with shared memory without bank conflicts

```
__global__ void transpose_lds_kernel(float *in, float *out, int width,
                                     int height) {
  __shared__ float tile[tile_dim][tile_dim+1];

  int x_tile_index = blockIdx.x * tile_dim;
  int y_tile_index = blockIdx.y * tile_dim;

  int in_index =
      (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.x);
  int out_index =
      (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.x);

  tile[threadIdx.y][threadIdx.x] = in[in_index];

  __syncthreads();

  out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```

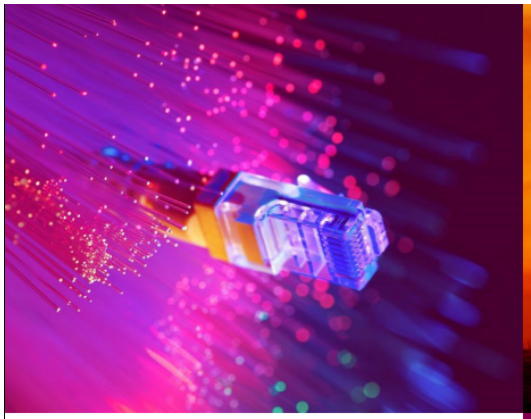
The duration is `0.179 ms` and the effective bandwidth `697 GB/s`

Other examples where shared memory is critical

- Matrix-matrix/vector multiplication
- N-body problem
- reductions

Summary

- Uses existing provided libraries: `hipBLAS`, `hipFFT`, ...
- Coalesced memory access in kernels results in better performance
- Use shared memory to reduce duplicate global memory accesses or make the acceses coalesced, but watch out for bank conflicts
- Try to avoid branching of the threads inside a wavefront



Multi-GPU programming and HIP+MPI

GPU programming with HIP

2022-11

CSC Training

CSC – Finnish expertise in ICT for research, education and public administration

Outline

- GPU context
- Device management
- Programming models
- Peer access (GPU-GPU)
- MPI+HIP

Introduction

- Workstations or supercomputer nodes can be equipped with several GPUs
  - For the current supercomputers, the number of GPUs per node usually ranges between 2 to 6
  - Allows sharing (and saving) resources (disks, power units, e.g.)
  - More GPU resources per node, better per-node-performance

GPU Context

- Context is established when the first HIP function requiring an active context is called (eg. `hipMalloc()`)
- Several processes can create contexts for a single device
- Resources are allocated per context
- By default, one context per device per process (since CUDA 4.0)
  - Threads of the same process share the primary context (for each device)

Selecting device

- Driver associates a number for each HIP-capable GPU starting from 0
- The function `hipSetDevice()` is used for selecting the desired device

Device management

Return the number of hip capable devices by `count`

```
hipError_t hipGetDeviceCount(int *count)
```

Set device as the current device for the calling host thread

```
hipError_t hipSetDevice(int device)
```

Return the current device for the calling host thread by `device`

```
hipError_t hipGetDevice(int *device)
```

Reset and explicitly destroy all resources associated with the current device

```
hipError_t hipDeviceReset(void)
```

Querying device properties

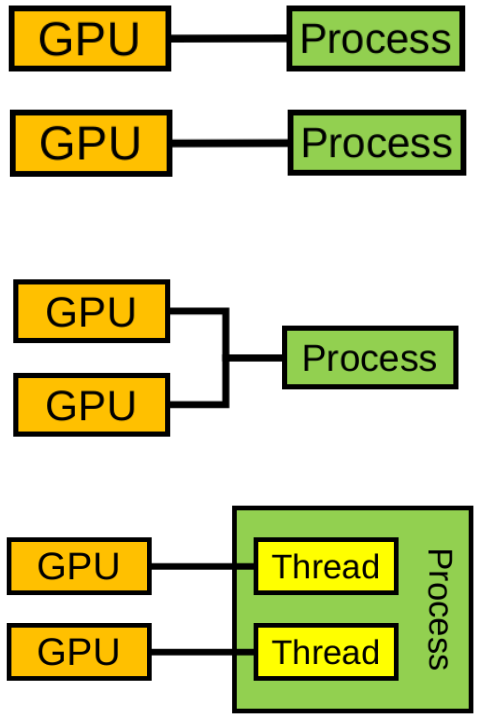
- One can query the properties of different devices in the system using `hipGetDeviceProperties()` function
  - No context needed
  - Provides e.g. name, amount of memory, warp size, support for unified virtual addressing, etc.
  - Useful for code portability

Return the properties of a HIP capable device by `prop`

```
hipError_t hipGetDeviceProperties(struct hipDeviceProp *prop, int device)
```

Multi-GPU programming models

- One GPU per process
  - Syncing is handled through message passing (eg. MPI)
- Many GPUs per process
  - Process manages all context switching and syncing explicitly
- One GPU per thread
  - Syncing is handled through thread synchronization requirements



Multi-GPU, one GPU per process

- Recommended for multi-process applications using a message passing library
- Message passing library takes care of all GPU-GPU communication
- Each process interacts with only one GPU which makes the implementation easier and less invasive (if MPI is used anyway)
  - Apart from each process selecting a different device, the implementation looks much like a single-GPU program
- **Multi-GPU implementation using MPI is discussed at the end!**

Multi-GPU, many GPUs per process

- Process switches the active GPU using `hipSetDevice()` function
- After setting the device, HIP-calls such as the following are effective only on the selected GPU:
  - Memory allocations and copies
  - Streams and events
  - Kernel calls
- Asynchronous calls are required to overlap work across all devices

Many GPUs per process, code example

```
for(unsigned int i = 0; i < deviceCount; i++)
{
  hipSetDevice(i);
  kernel<<<blocks[i],threads[i]>>>(arg1[i], arg2[i]);
}
```

Multi-GPU, one GPU per thread

- One GPU per CPU thread
  - I.e one OpenMP thread per GPU being used
- HIP API is threadsafe
  - Multiple threads can call the functions at the same time
- Each thread can create its own context on a different GPU
  - `hipSetDevice()` sets the device and creates a context per thread
  - Easy device management with no changing of device
- Communication between threads becomes a bit more tricky

One GPU per thread, code example

```
#pragma omp parallel for
for(unsigned int i = 0; i < deviceCount; i++)
{
  hipSetDevice(i);
  kernel<<<blocks[i],threads[i]>>>(arg1[i], arg2[i]);
}
```

Peer access

- Access peer GPU memory directly from another GPU
  - Pass a pointer to data on GPU 1 to a kernel running on GPU 0
  - Transfer data between GPUs without going through host memory
  - Lower latency, higher bandwidth

```
// Check peer accessibility
hipError_t hipDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice)

// Enable peer access
hipError_t hipDeviceEnablePeerAccess(int peerDevice, unsigned int flags)

// Disable peer access
hipError_t hipDeviceDisablePeerAccess(int peerDevice)
```

Peer to peer communication

- Devices have separate memories
- With devices supporting unified virtual addressing, `hipMemCpy()` with `kind=hipMemcpyDefault`, works:

```
hipError_t hipMemcpy(void* dst, void* src, size_t count, hipMemcpyKind kind)
```

- Other option which does not require unified virtual addressing

```
hipError_t hipMemcpyPeer(void* dst, int  dstDev, void* src, int srcDev, size_t count
```

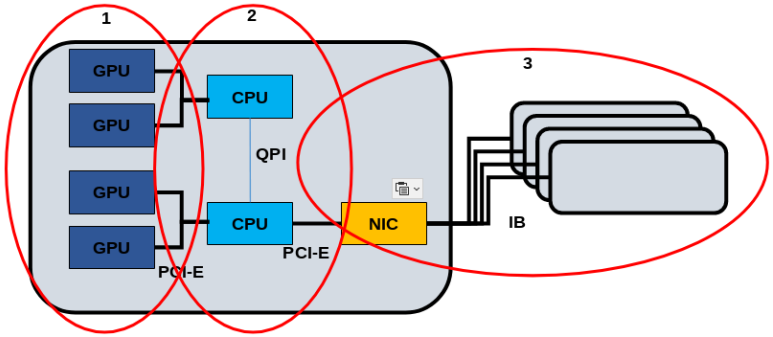
- If peer to peer access is not available, the functions result in a normal copy through host memory

Message Passing Interface (MPI)

- MPI is a widely adopted standardized message passing interface for distributed memory parallel computing
- The parallel program is launched as a set of independent, identical processes
  - Same program code and instructions
  - Each process can reside in different nodes or even different computers
- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages

Three levels of parallelism

1. GPU - GPU threads on the multiprocessors
    - Parallelization strategy: HIP, SYCL, Kokkos, OpenMP
2. Node - Multiple GPUs and CPUs
    - Parallelization strategy: MPI, Threads, OpenMP
3. Supercomputer - Many nodes connected with interconnect
    - Parallelization strategy: MPI between nodes



MPI and HIP

- Trying to compile code with any HIP calls with other than the `hipcc` compiler can result in errors
- Either set MPI compiler to use `hipcc`, eg for OpenMPI:

```
OMPI_CXXFLAGS='' OMPI_CXX='hipcc'
```

- or separate HIP and MPI code in different compilation units compiled with `mpicxx` and `hipcc`
  - Link object files in a separate step using `mpicxx` or `hipcc`

MPI+HIP strategies

1. One MPI process per node
2. **One MPI process per GPU**
3. Many MPI processes per GPU, only one uses it
4. **Many MPI processes sharing a GPU**

- 2 is recommended (also allows using 4 with services such as CUDA MPS)
  - Typically results in most productive and least invasive implementation for an MPI program
  - No need to implement GPU-GPU transfers explicitly (MPI handles all this)
  - It is further possible to utilize remaining CPU cores with OpenMP (but this is not always worth the effort/increased complexity)

Selecting the correct GPU

- Typically all processes on the node can access all GPUs of that node
- The following implementation allows utilizing all GPUs using one or more processes per GPU
  - Use CUDA MPS when launching more processes than GPUs

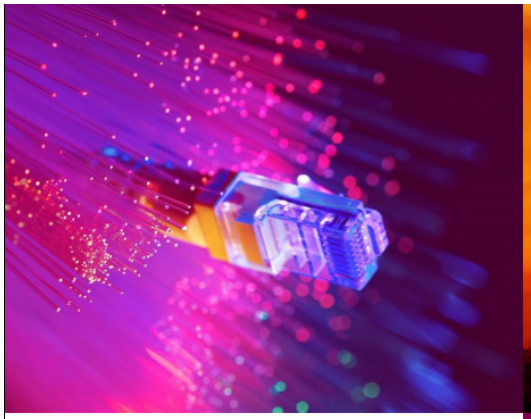
```
int deviceCount, nodeRank;
MPI_Comm commNode;
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &commNod
MPI_Comm_rank(commNode, &nodeRank);
hipGetDeviceCount(&deviceCount);
hipSetDevice(nodeRank % deviceCount);
```

GPU-GPU communication through MPI

- CUDA/ROCm aware MPI libraries support direct GPU-GPU transfers
  - Can take a pointer to device buffer (avoids host/device data copies)
- Unfortunately, currently no GPU support for custom MPI datatypes (must use a datatype representing a contiguous block of memory)
  - Data packing/unpacking must be implemented application-side on GPU
- ROCm aware MPI libraries are under development and there may be problems
  - It is a good idea to have a fallback option to use pinned host staging buffers

Summary

- There are many options to write a multi-GPU program
- Use `hipSetDevice()` to select the device, and the subsequent HIP calls operate on that device
- If you have an MPI program, it is often best to use one GPU per process, and let MPI handle data transfers between GPUs
- There is still little experience from ROCm aware MPIs, there may be issues
  - Note that a CUDA/ROCm aware MPI is only required when passing device pointers to the MPI, passing only host pointers does not require any CUDA/ROCm awareness



Code design, conditional compilation, lambdas, hipify

GPU programming with HIP

2022-11

CSC Training

CSC – Finnish expertise in ICT for research, education and public administration

Outline

- Introduction
- Conditional compilation
- General kernels with lambda functions
- Memory management
- Hipify-tools - translating CUDA to HIP
- Error checking reminder

Introduction

- Ok, now we know some HIP functions, so how to apply this knowledge to real projects?
- How to separate the architecture specific code from the numerical methods?
- How to make HIP an optional feature, and allow my code to still compile with plain C compiler?
- Do I need to rewrite all 150 loops in my huge codebase with a device kernel for each?

Conditional compilation

- Many projects find it desirable to add HIP as an optional dependence
- However, this can lead lead to a messy code base which is difficult to maintain (see the example ->)
  - The issue can be avoided by separating the architecture and the numerical implementation (separation of concerns)
- For example, let's try putting HIP code in arch_hip.h and the respective non-HIP code in arch_host.h (see the next slides)
  - Architecture-specific code is kept separately in one place
  - This approach works very nicely with generic GPU kernels that exploit lambda functions

```
#if defined(HAVE_HIP)
  #include <hip/hip_runtime.h>
#endif

int main(){
  int* ptr;
  #if defined(HAVE_HIP)
    hipMallocManaged(&ptr, 1024);
  #else
    ptr = malloc(1024);
  #endif
  /* Do something here*/
  #if defined(HAVE_HIP)
    /* Run GPU loop (kernel) */
  #else
    /* Run CPU loop */
  #endif
  /* Do something here */
  #if defined(HAVE_HIP)
    hipFree(ptr);
  #else
    free(ptr);
  #endif
}
```

Conditional compilation with separation of concerns

- .cpp file

```
/* HAVE_DEF determines which accelerator backend
 * is used (and is checked only ONCE) */
#if defined(HAVE_HIP)
  #include "arch_hip.h"
#else
  #include "arch_host.h"
#endif

/* The main function */
int main(){

  /* Memory allocation depends on the attached
   * header, ie, the chosen architecture */
  int* array = (int*) arch::alloc(1024);

  /* Here do something with the array */

  /* Memory deallocation depends on the attached
   * header, ie, the chosen architecture */
  arch::dealloc(array);
}
```

- .h files (headers)

```
/* arch_hip.h */
#include <hip/hip_runtime.h>
namespace arch{
  static void* alloc(size_t bytes) {
    void* ptr;
    hipMallocManaged(&ptr, bytes);
    return ptr;
  }
  static void dealloc(void* ptr) {
    hipFree(ptr);
  }
}
```

```
/* arch_host.h */
namespace arch{
  static void* alloc(size_t bytes) {
    return malloc(bytes);
  }
  static void dealloc(void* ptr) {
    free(ptr);
  }
}
```

General GPU kernels - Lambda function

- In C++, lambda functions can be used to create general GPU kernels
- Lambda function is a function object which can be passed for other functions as an argument
- Lambda function can “capture” other variables by reference [&] or by value [=] from the scope where it is defined
- Lambda functions work nicely with the conditional compilation approach
  - The loop contents can be put into a lambda function, which are then executed by the chosen method (eg, a HIP kernel or a sequential for-loop running on host)

- Lambda function example with capture

```
#include <stdio.h>
int main() {
  int i = -9;
  /* Define a lambda function which captures the
   * required variables by their value, ie, [=] */
  auto lambda = [=](int j) {
    printf("i = %d (captured value), j = %d (passed value)
  };
  /* Call the lambda function in a loop */
  for(i = 0; i < 3; i++)
    lambda(i);
  return 0;
}
```

```
Output:
i = -9 (captured value), j = 0 (passed value)
i = -9 (captured value), j = 1 (passed value)
i = -9 (captured value), j = 2 (passed value)
```

General GPU kernels - Device execution

- .cpp file

```
#if defined(HAVE_HIP)
  #include "arch_hip.h"
#else
  #include "arch_host.h"
#endif

int main(){
  int n_array = 10;
  int* array = (int*) arch::alloc(n_array * sizeof(int));
  /* Run on host or on a device depending
   * on the chosen compile configuration */
  arch::parallel_for(n_array,
    LAMBDA(const uint i) {
      array[i] *= 2;
    }
  );
  arch::dealloc(array);
}
```

- Lambda function can be passed to a GPU kernel by adding `__host__ __device__` identifiers, ie, by making it host-device lambda function

- .h file (device header)

```
/* arch_hip.h */
#include <hip/hip_runtime.h>
#define LAMBDA [=] __host__ __device__

namespace arch{
  static void* alloc(size_t bytes) { /* definition not sho
  static void dealloc(void* ptr) { /* definition not shown

  template <typename T>
  __global__ static void hipKernel(T lambda, const uint lo
    const uint i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < loop_size){
      lambda(i);
    }
  }
  template <typename T>
  static void parallel_for(uint loop_size, T loop_body){
    const uint blocksize = 64;
    const uint gridsize = (loop_size - 1 + blocksize) / bl
    hipKernel<<<gridsize, blocksize>>>(loop_body, loop_siz
    hipStreamSynchronize(0);
  }
}
```

General GPU kernels - Host execution

- .cpp file

```
#if defined(HAVE_HIP)
  #include "arch_hip.h"
#else
  #include "arch_host.h"
#endif

int main(){
  int n_array = 10;
  int* array = (int*) arch::alloc(n_array * sizeof(int));
  /* Run on host or on a device depending
   * on the chosen compile configuration */
  arch::parallel_for(n_array,
    LAMBDA(const uint i) {
      array[i] *= 2;
    }
  );
  arch::dealloc(array);
}
```

- .h file (host header)

```
/* arch_host.h */
#define LAMBDA [=]

namespace arch{
  static void* alloc(size_t bytes) { /* definition not sho
  static void dealloc(void* ptr) { /* definition not shown

  template <typename T>
  static void parallel_for(uint loop_size, T loop_body) {
    for(uint i = 0; i < loop_size; i++){
      loop_body(i);
    }
  }
}
```

- `__host__ __device__` identifiers are not used when compiled for host execution

Support for plain C interface through macros

- .cpp file

```
#if defined(HAVE_HIP)
  #include "arch_hip.h"
#else
  #include "arch_host.h"
#endif

int main(){
  int n_array = 10;
  int* array = (int*) arch_alloc(n_array * sizeof(int));
  /* Run on host or on a device depending
   * on the chosen compile configuration */
  int i;
  arch_parallel_for(n_array, i,
  {
      array[i] *= 2;
  });
  arch_dealloc(array);
}
```

- The interface preserves compatibility with C compilers when compiled without HIP
  - May be desirable for ported legacy C codes

- .h file (device header)

```
/* arch_hip.h */
#include <hip/hip_runtime.h>
#define LAMBDA [=] __host__ __device__

static void* alloc(size_t bytes) { /* definition not shown
static void dealloc(void* ptr) { /* definition not shown *

template <typename T>
__global__ static void hipKernel(T lambda, const uint loop_
  const uint i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < loop_size){
    lambda(i);
  }
}

#define arch_parallel_for(loop_size, inc, loop_body)
{
  const int blocksize = 64;
  const int gridsize = (loop_size - 1 + blocksize) / block
  auto lambda_body = LAMBDA (int inc) loop_body;
  hipKernel<<<gridsize, blocksize>>>(lambda_body, loop_siz
  hipStreamSynchronize(0);
}
```

Memory management - Unified memory

- Using unified memory is simple and easy, and thus a good candidate to begin the porting effort
- Memory transfers are triggered automatically by a page-fault mechanism
- The allocations and frees (eg, `malloc()` and `free()`) are simply replaced by `hipMallocManaged()` and `hipFree()`
- Works with AMD and NVIDIA architectures
- However, large amount of recurring page faults may lead to a big performance penalty
- Also, support for unified memory is not guaranteed across all architectures (especially older ones)

```
/* arch_hip.h */
#include <hip/hip_runtime.h>
namespace arch{
  static void* alloc(size_t bytes) {
    void* ptr;
    hipMallocManaged(&ptr, bytes);
    return ptr;
  }
  static void dealloc(void* ptr) {
    hipFree(ptr);
  }
}
```

```
/* arch_host.h */
namespace arch{
  static void* alloc(size_t bytes) {
    return malloc(bytes);
  }
  static void dealloc(void* ptr) {
    free(ptr);
  }
}
```

Memory management - Buffers

- Create data classes in arch_hip.c and arch_host.h, which take care of the memory operations
  - Can be tailored for the underlying application
  - Look into Kokkos “View” and Sycl “buffer” for ideas

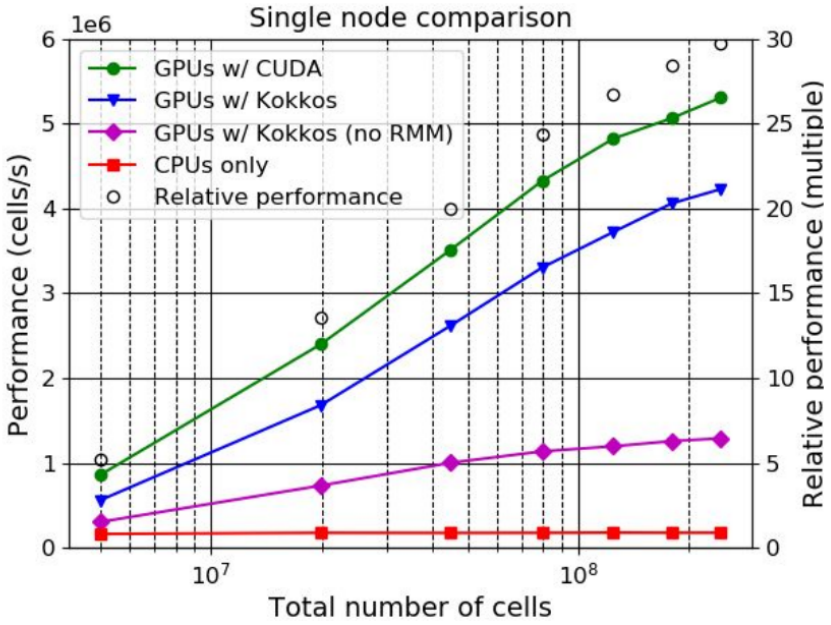
```
#include "arch_hip.h"

int main(){
  int n_array = 10;
  int* host_array = (int*) malloc(n_array * sizeof(int));

  {
    /* The constructor of Buf class copies data to device
    arch::Buf array(host_array, n_array * sizeof(int));
    /* The data accessed from the kernel */
    arch::parallel_for(n_array,
      LAMBDA(const uint i) {
        array[i] *= 2;
      }
    );
   /* The destructor of Buf class copies data back to host
  }
  free(host_array);
}
```

```
/* arch_hip.h */
class Buf {
  T *ptr, *d_ptr;
  uint bytes, is_copy = 0;

  public:
  Buf(int *_ptr, uint _bytes) : ptr(_ptr), bytes(_bytes) {
    hipMalloc(&d_ptr, bytes);
    hipMemcpy(d_ptr, ptr, bytes, hipMemcpyHostToDevice);
  }
  __host__ __device__ Buf(const Buf& u) :
    ptr(u.ptr), d_ptr(u.d_ptr), bytes(u.bytes), is_copy(1)
  __host__ __device__ ~Buf(void){
    if(!is_copy){
      hipMemcpy(ptr, d_ptr, bytes, hipMemcpyDeviceToHost);
      hipFree(d_ptr);
  } }
  __host__ __device__ int &operator [] (uint i) const {
   #ifdef __HIP_DEVICE_COMPILE__
      return d_ptr[i]; /* if accessed from device code */
   #else
      return NULL; /* illegal access when accessed from ho
   #endif
  }
};
```

Real world example: ParFlow

- 3D model for variably saturated subsurface flow including pumping and irrigation, and integrated overland flow
- Integrated with land surface and atmospheric models
- FD/FV schemes, implicit Newton-Krylov multigrid preconditioned solver
- Dozens of numerical kernels, no single hotspot, 150k lines of C
- Ported to GPUs using unified memory + general device kernels
  - The interface use macros to maintain compatibility with C compilers (when compiled without GPU support)
  - Supports four backend options, sequential CPU, OpenMP, CUDA, and Kokkos

- Up to 30x speedup over CPU version
- Good weak scaling across +1000 GPUs
- Unified Memory pool allocator (RMM) makes a huge difference!



Hipify-tools - translating CUDA to HIP

- Hipify-tools can translate CUDA source code into portable HIP C++ automatically
- Although most CUDA expressions are supported, manual intervention may be required
  - For example, a CUDA macro `__CUDA_ARCH__` is not translated
    - If the purpose of `__CUDA_ARCH__` is to distinguish between host and device code path, it can be replaced with `__HIP_DEVICE_COMPILE__`
    - If `__CUDA_ARCH__` is used to determine architectural feature support, another solution is required, eg, `__HIP_ARCH_HAS_DOUBLES__`

Hipify-tools - translating CUDA to HIP

- To access Hipify-tools on Puhti, do:

```
ml purge; ml gcc/11.3.0 hipify-clang/5.1.0
```

- hipify-clang: CUDA -> HIP translator based on LLVM clang
  - Only syntactically correct CUDA code is translated
  - Good support even for somewhat complicated constructs
  - Requires third party dependencies:
    - 3.8.0 <= clang <= 13.0.1
    - 7.0 <= CUDA <= 11.5.1
  - Usage (-print-stats is optional, but on Puhti, –cuda-path must be specified):

```
hipify-clang -print-stats -o src.cu.hip src.cu --cuda-path=/appl/spack/v018/install-tree/gcc-9.4.0/cuda-1
```

- hipify-perl: a perl script for CUDA -> HIP translation that mostly uses regular expressions
  - Does not check the input CUDA code for correctness
  - No third party dependencies like clang or CUDA
  - Not as reliable as hipify-clang
  - Usage (-print-stats is optional):

```
hipify-perl -print-stats -o src.cu.hip src.cu
```

Error checking reminder

- Always use HIP error checking!
  - It has low overhead, and can save a lot of debugging time!

```
#define HIP_ERR(err) (hip_error(err, __FILE__, __LINE__))
inline static void hip_error(hipError_t err, const char *file, int line) {
  if (err != hipSuccess) {
    printf("\n\n%s in %s at line %d\n", hipGetErrorString(err), file, line);
    exit(1);
  }
}
```

- Here we wrap a hip call into the above defined HIP_ERR macro:

```
inline static void* alloc(size_t bytes) {
  void* ptr;
  HIP_ERR(hipMallocManaged(&ptr, bytes));
  return ptr;
}
```

Summary

- A nice way to make HIP an optional dependence, is to create a unified interface for looping and memory management
  - The definitions for the interface are placed in architecture specific headers
  - Using macros instead of functions allows the interface to be compatible with C and legacy codes
- Starting a porting effort with unified memory might be a good idea, because it is trivial
  - Changing to explicitly managed memory is possible later if needed
- Using general device kernels (with lambdas) can reduce code duplication
- Always using HIP error checking is strongly recommended

facebook.com/CSCfi

twitter.com/CSCfi

linkedin.com/company/csc---it-center-for-science

github.com/CSCfi