Hewlett Packard
Enterprise

Introductory Programming of AMD GPUs with HIP

Harvey Richardson, HPE HPC&AI EMEA Research Lab

PDC Summer School, Aug 21-22, 2024

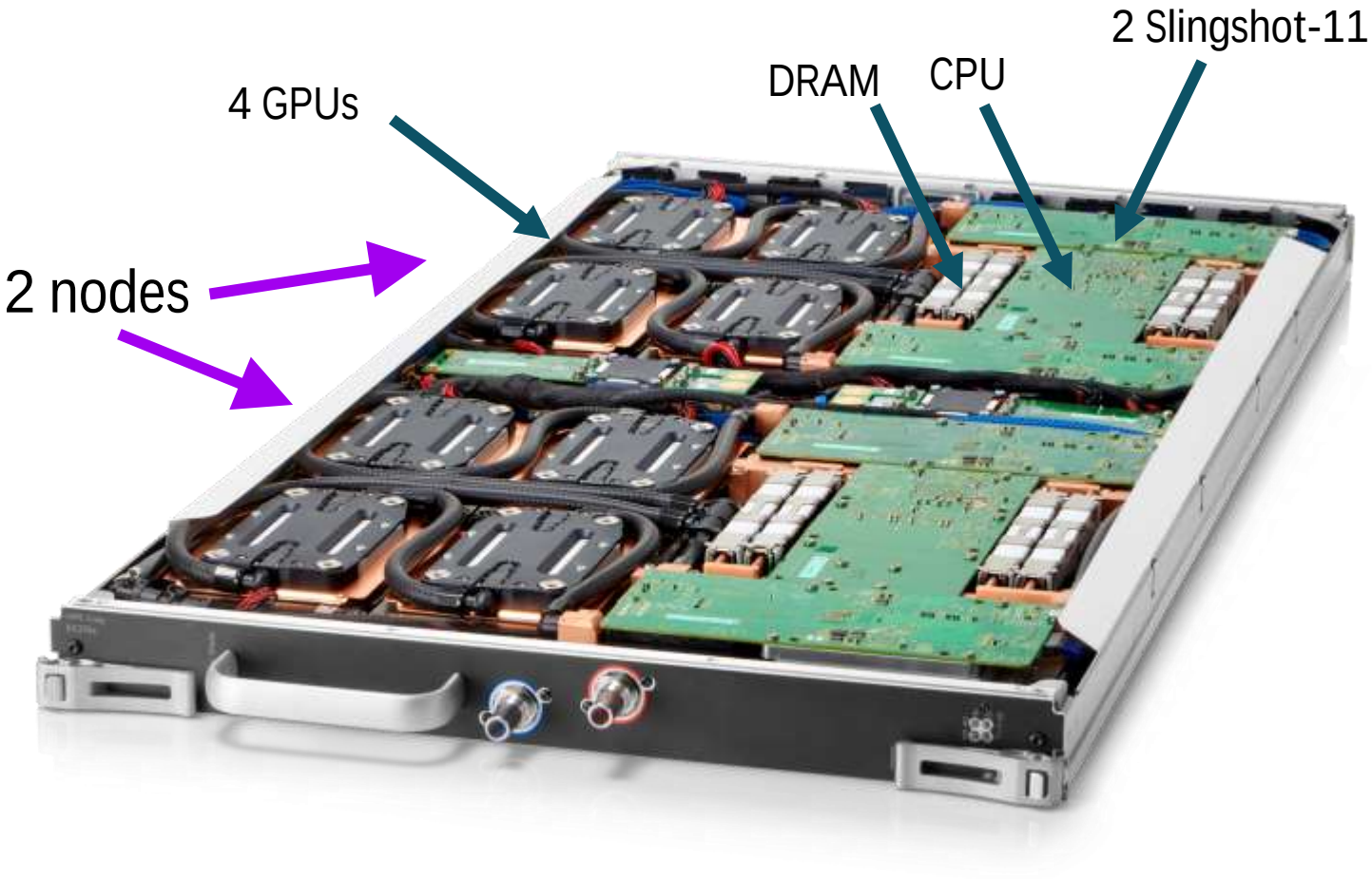
© Copyright 2024 Hewlett Packard Enterprise Development LP

Agenda

- Simple generic CPU/GPU architecture
- The basics of programming in HIP
- First steps, use the HIP API to detect GPUs
- Fundamental parts of a HIP application
  - Memory allocation
  - Copying data to the GPU
  - Executing a workload on the GPU
  - Copying data from the GPU
- Mapping workloads to the GPU and how hardware comes into this
- API comparison with CUDA
- Other considerations

© Copyright 2024 Hewlett Packard Enterprise Development LP

How do we program this ?

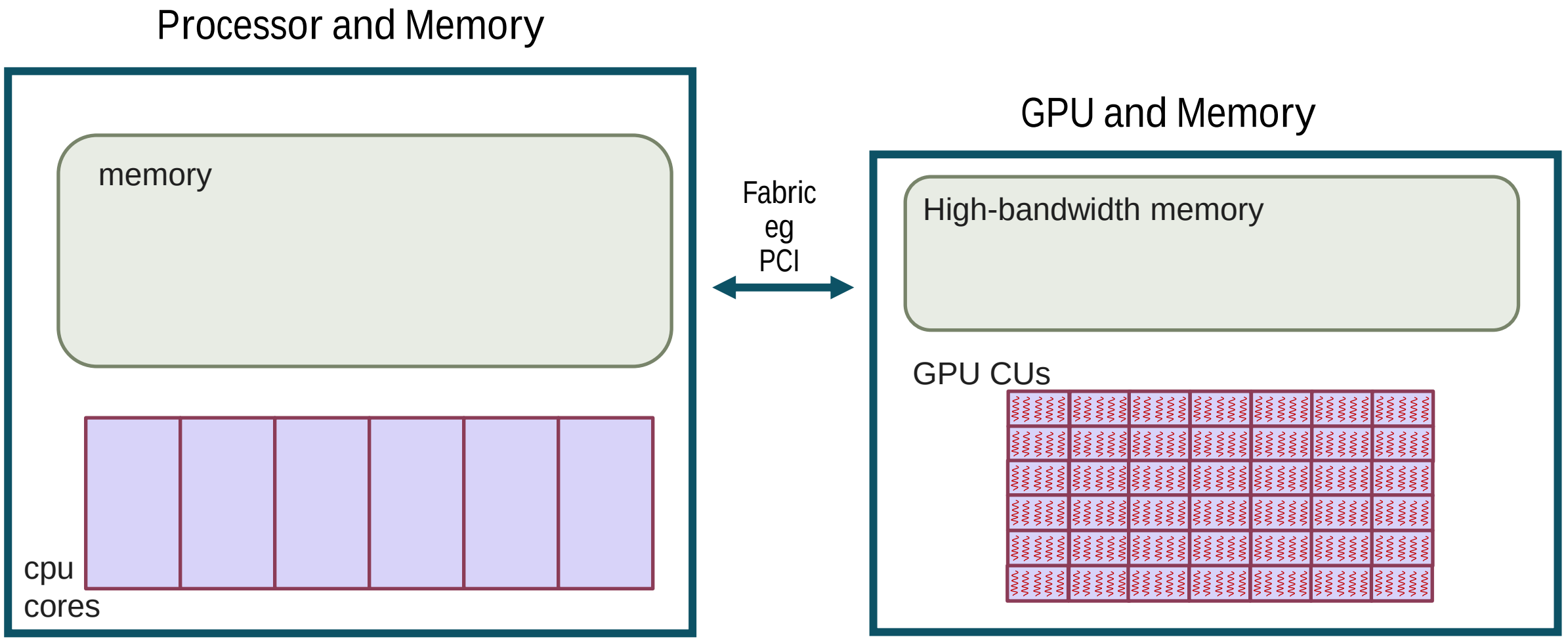


Each node (LUMI/DARDEL):

- AMD EPYC 7A53 “Optimized 3rd Gen EPYC” 64-Core Processor, 2.00 GHz
- 512 GB DDR4 memory
- 4x AMD MI-250X GPU each with 128GB HBM2e each with 220 CUs
- Each GPU connected to a Slingshot 200Gb/s NIC

© Copyright 2024 Hewlett Packard Enterprise Development LP

Consider the simple architecture



© Copyright 2024 Hewlett Packard Enterprise Development LP

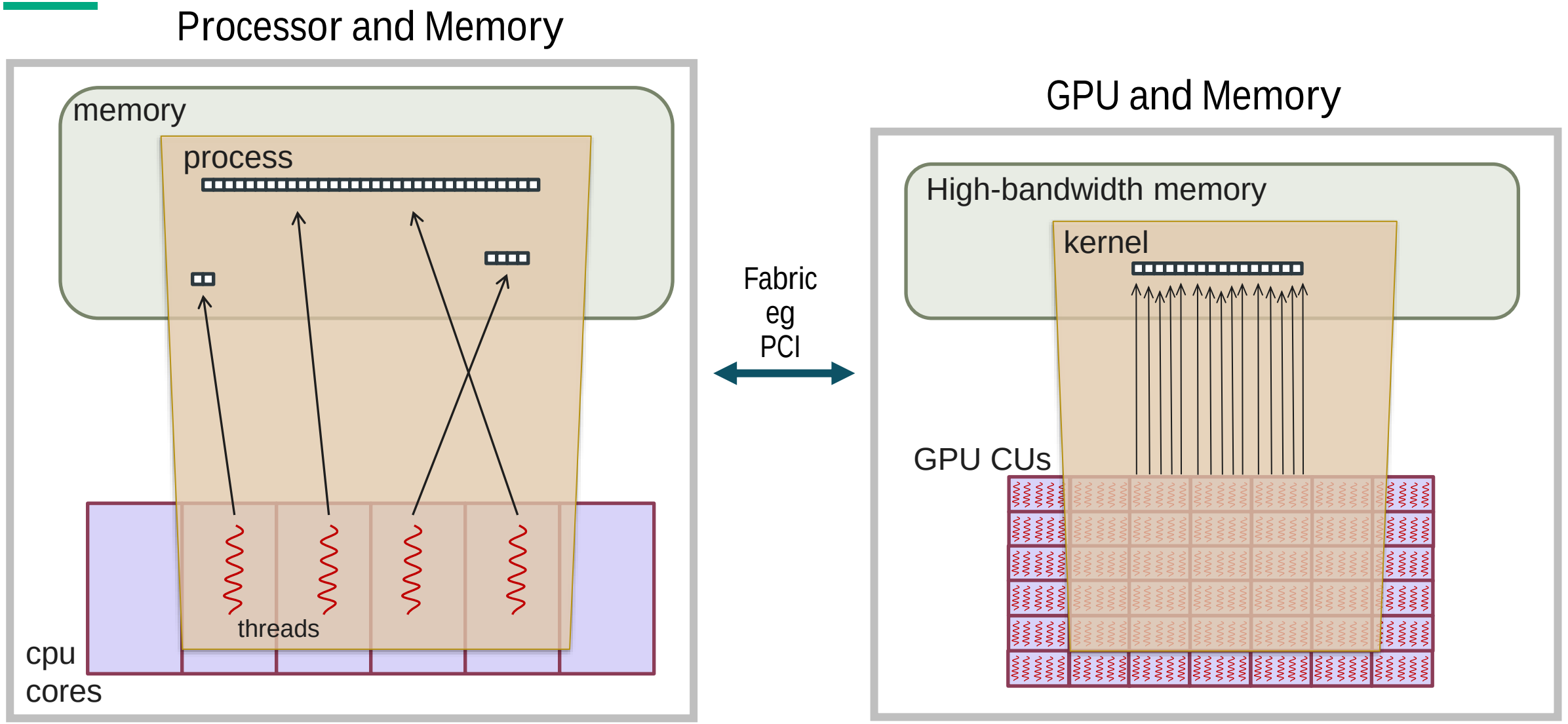
GPU programming approaches from NVIDIA and AMD

- GPUs (Graphics Processing Units) were programmed by graphics developers using specific APIs
- NVIDIA introduced the Compute Unified Device Architecture (CUDA) programming model in 2006 which enabled more general-purpose programming of GPUs. Subsequent support for 64-bit floating-point arithmetic and ECC memory support (2009) made GPUs attractive to a wider user base
- The CUDA programming model has the essential components needed to integrate a GPU into a scientific program
- Main CUDA features
  - Language based on C++
  - GPU Hardware abstractions: hierarchy of thread groups, shared memories and synchronisation

- AMD Introduced the C++ Heterogeneous-Compute Interface for Portability (HIP) as part of the ROCm compute software stack for AMD GPUs
  - HIP is very similar to CUDA and often little more than a search-replace in the API is all that is needed to convert from CUDA to HIP

© Copyright 2024 Hewlett Packard Enterprise Development LP

Consider the simple architecture if we want to program it



© Copyright 2024 Hewlett Packard Enterprise Development LP

The basics of programming in HIP

- HIP provides a runtime API and a C++ programming language
- HIP allows us to run a program on a host which launches computational kernels on a target device
- HIP is a single-source programming model
- The context is of a single thread running on the host targeting a GPU device

The fundamentals for a simple nontrivial HIP program:

- Allocate data in host or device memories
- Transfer data between host and device
- Launch computational kernels that execute on the device and which operate on a large number of blocks or work groups of threads
  - We often call this offloading to the GPU
- Transfer data (results) from device to host

© Copyright 2024 Hewlett Packard Enterprise Development LP

A very simple HIP program to enquire about GPUs

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

int main(int argc, char **argv){
  int num_devices=0;
  hipError_t herr;
  hipDeviceProp_t props;

  herr = hipGetDeviceCount(&num_devices);
  if (herr != hipSuccess){ num_devices=-1;}
   for(int i=0; i<num_devices; i++){
     hipGetDeviceProperties(&props, i);
     printf("Device %d: %d CUs, %zuMB\n",
            i, props.multiProcessorCount, props.totalGlobalMem/(1<<20));
   }
  return EXIT_SUCCESS;
}
```

Note error code / status

- Output:

```
Device 0: 110 CUs, 65520MB
Device 1: 110 CUs, 65520MB
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

Checking Errors

- HIP API routines typically return an error-code/status which is saved
- Make it a habit to check these
- To check last error use `hipGetLastError(e)`
- To check last error without resetting saved status use `hipPeekLastError(e)`
- To get error name and description use `hipGetErrorName(e),hipGetErrorString(e)`

In many examples you will see use of macros to test the last error or check at a call site, for example:

```
#define HIP_CHECK (hip_api_call) { \
  hipError_t herr = hip_api_call; \
  if( herr ) { \
  std::cerr << "Error(HIP): " << hipGetErrorString(herr) << endl ; \
std::abort (); } }
```

- An example error report might be “Error(HIP): hipErrorInvalidDevice”
- For brevity we will not include error checking in subsequent slides

© Copyright 2024 Hewlett Packard Enterprise Development LP

Allocating device memory

- We need to use the HIP API to allocate device memory
- For example, to allocate a vector of N doubles on the device:

```
double *d_v = NULL;
size_t v_nbytes = N * sizeof(double);

hipMalloc(&d_v, v_nbytes);
```

- It is a common convention to use h (host) and d (device) in variable names to differentiate host and device pointers.
- To free device memory:

```
hipFree( d_v );
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

Moving data between host and device

- Assume a same-sized vector allocated on the host

```
double *v = (double *) malloc( (size_t) N * sizeof(double));
```

- We can copy the vector from host to device with

```
hipMemcpy(d_v, v, v_nbytes, hipMemcpyHostToDevice);
```

- And to copy back from device to host

```
hipMemcpy(v, d_v, v_nbytes, hipMemcpyDeviceToHost);
```

- Device to device copies can use `hipMemcpy` with `hipMemcpyDeviceToDevice`

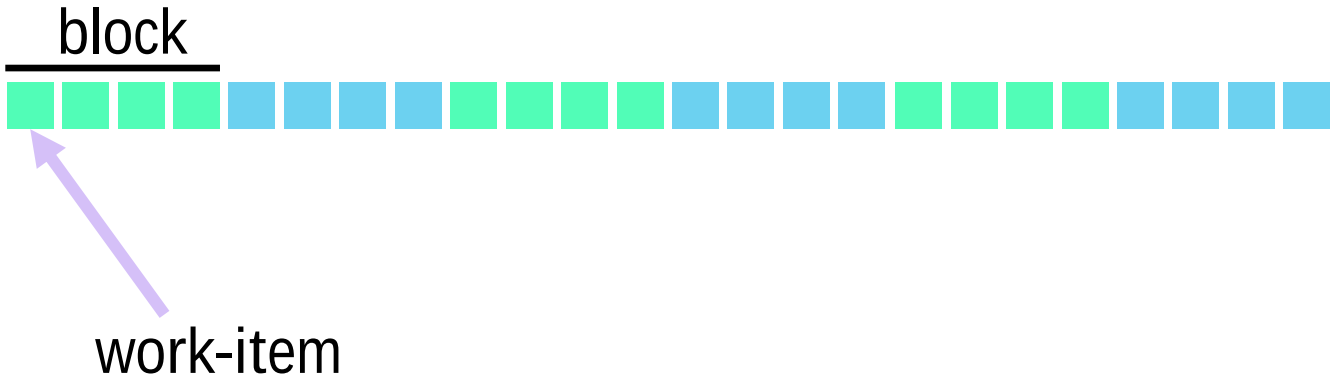
© Copyright 2024 Hewlett Packard Enterprise Development LP

The HIP approach to expressing parallelism

- Now we know how to get data onto the device we need some way to describe how to operate on that data in parallel.
- HIP uses a grid model for this
  - At the lowest level we describe a block of work items (sometimes called threads)
  - Each work-item is indexed into a 3-dimensional space (1-d,2-d supported)
  - Blocks are organised into a grid

© Copyright 2024 Hewlett Packard Enterprise Development LP

Grid model in 1-D

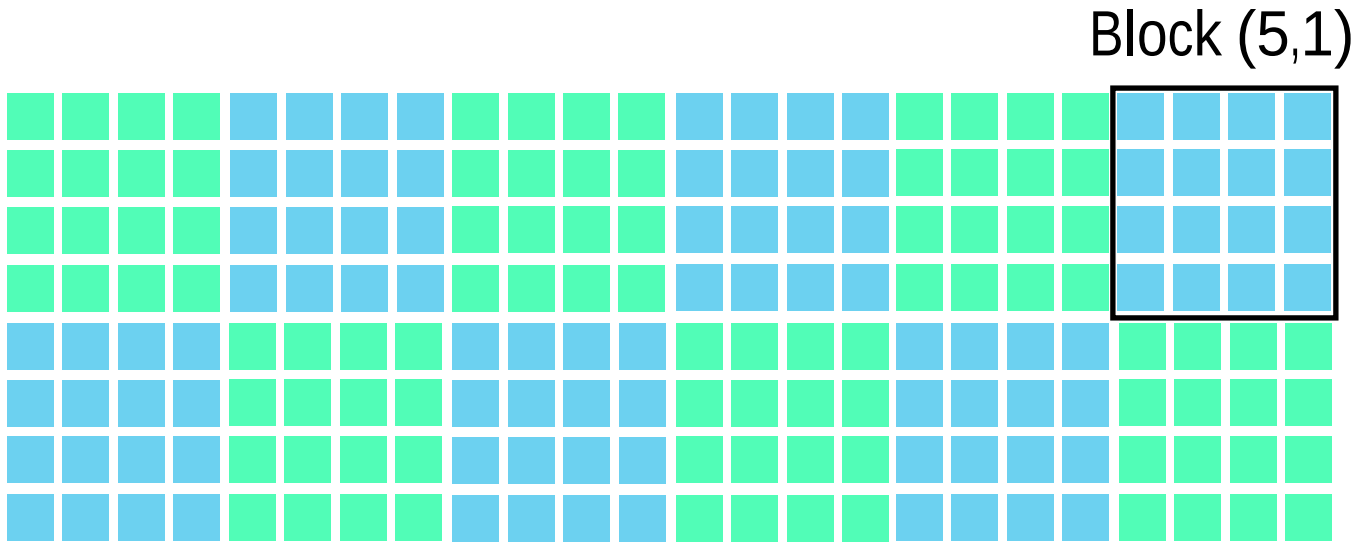


Once the grid details are defined HIP allows us to access these via *coordinate built-ins* follows:

- Each block is of size blockDim.x
- Each work-item in a block has index threadIdx.x
- Each block in the grid has an index blockIdx.x
- The dimension of the grid (number of blocks) is gridDim.x
- Note that gridDim.x * blockDim.x must be < 2^32

© Copyright 2024 Hewlett Packard Enterprise Development LP

Grid model in 2-D



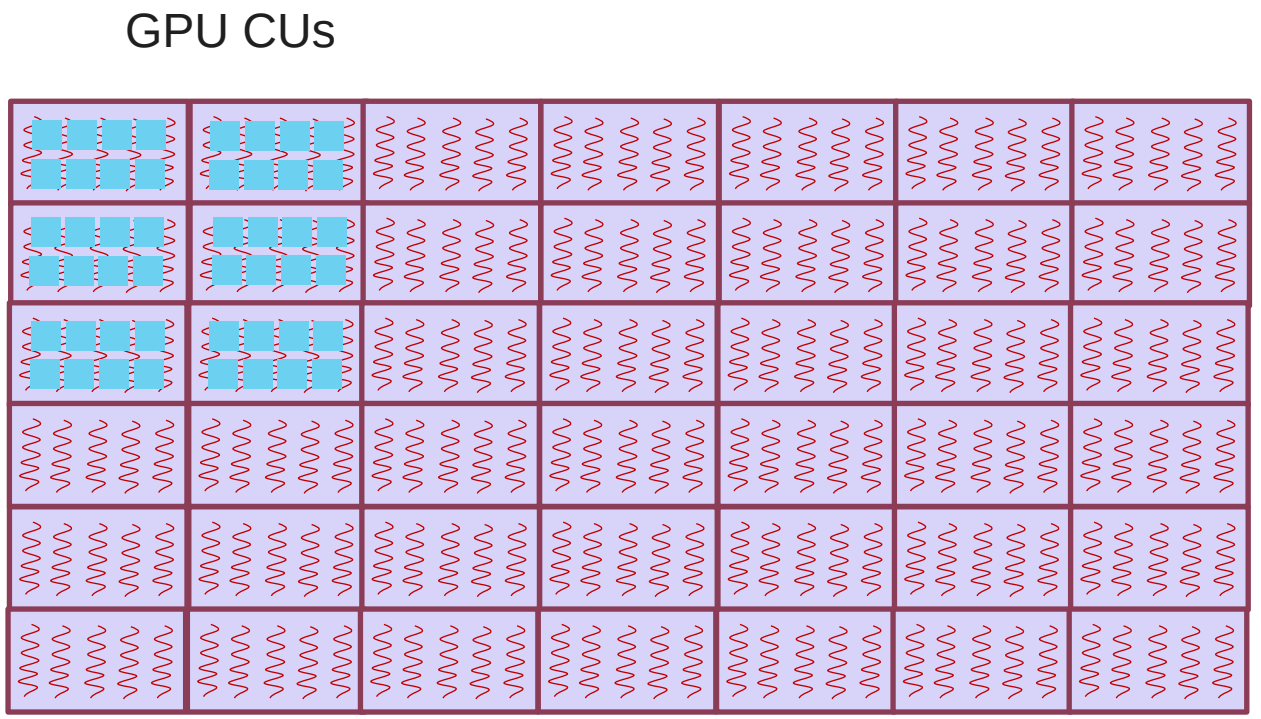
Extending to 3-D we have .x, .y, .z so for above

- Each block is of size blockDim.x x BlockDim.y (4,4,1)
- Each work-item in a block has index threadIdx.x, threadIdx.y
- Each block in the grid has an index blockIdx.x, blockIdx.y
- The dimension of the grid (in blocks) is gridDim.x, gridDim.y, gridDim.z (6,2,1)

© Copyright 2024 Hewlett Packard Enterprise Development LP

How do we size the grid?

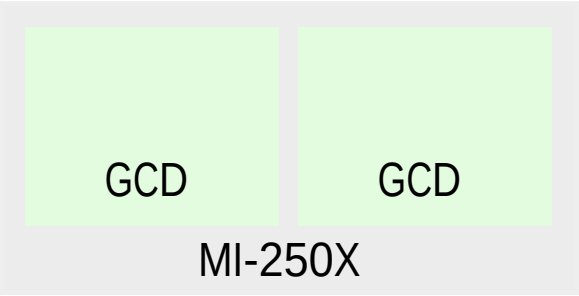
- We need to consider how work is mapped to the GPU
- Blocks are scheduled to Compute Units (CUs)
- Each CU contains SIMD hardware that can execute instructions relating to many work-items at once
- We need to know more about hardware to answer this question…



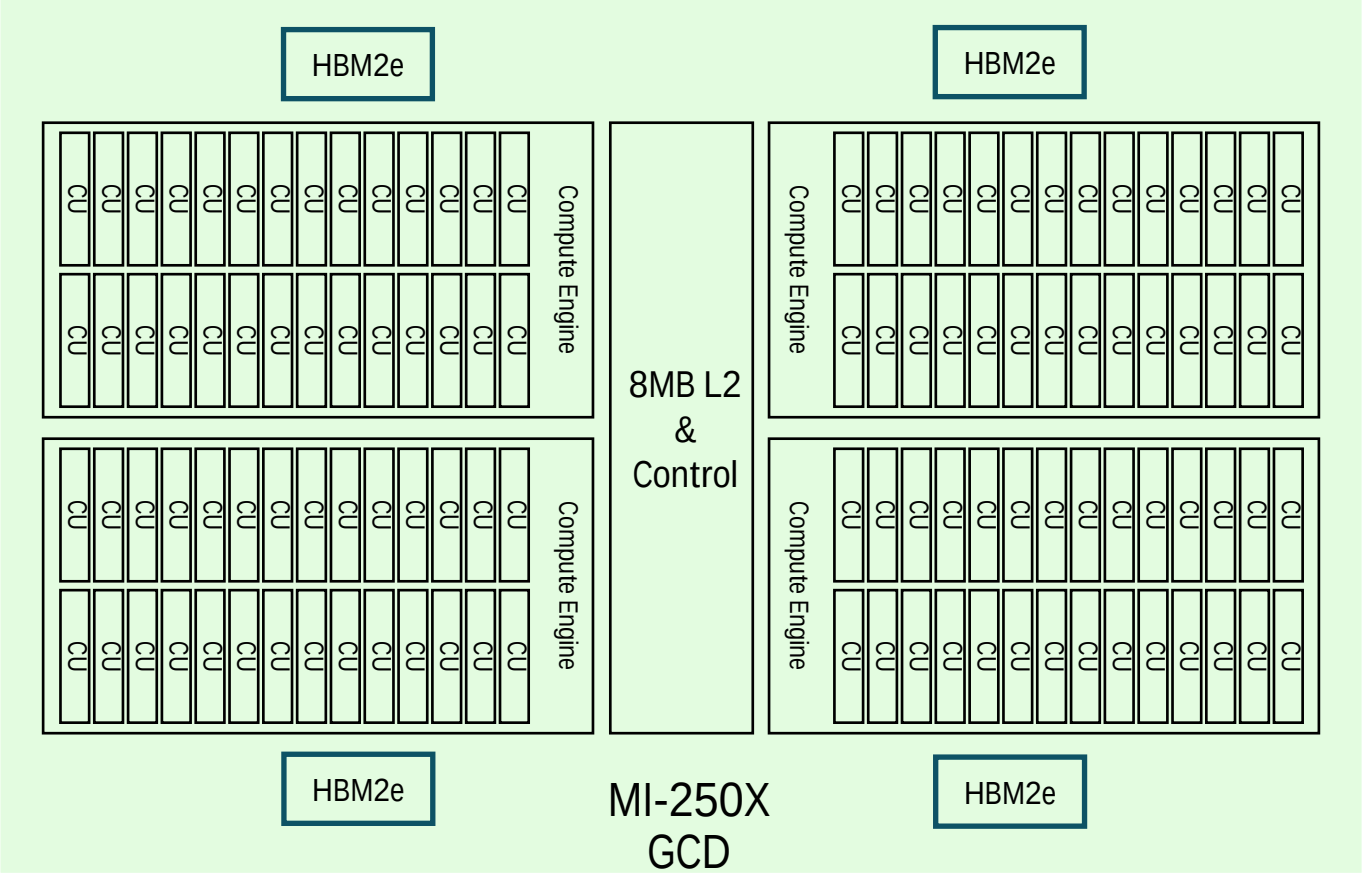
© Copyright 2024 Hewlett Packard Enterprise Development LP

Consider real hardware (AMD MI250X)

- The MI250X package includes 2x GCDs

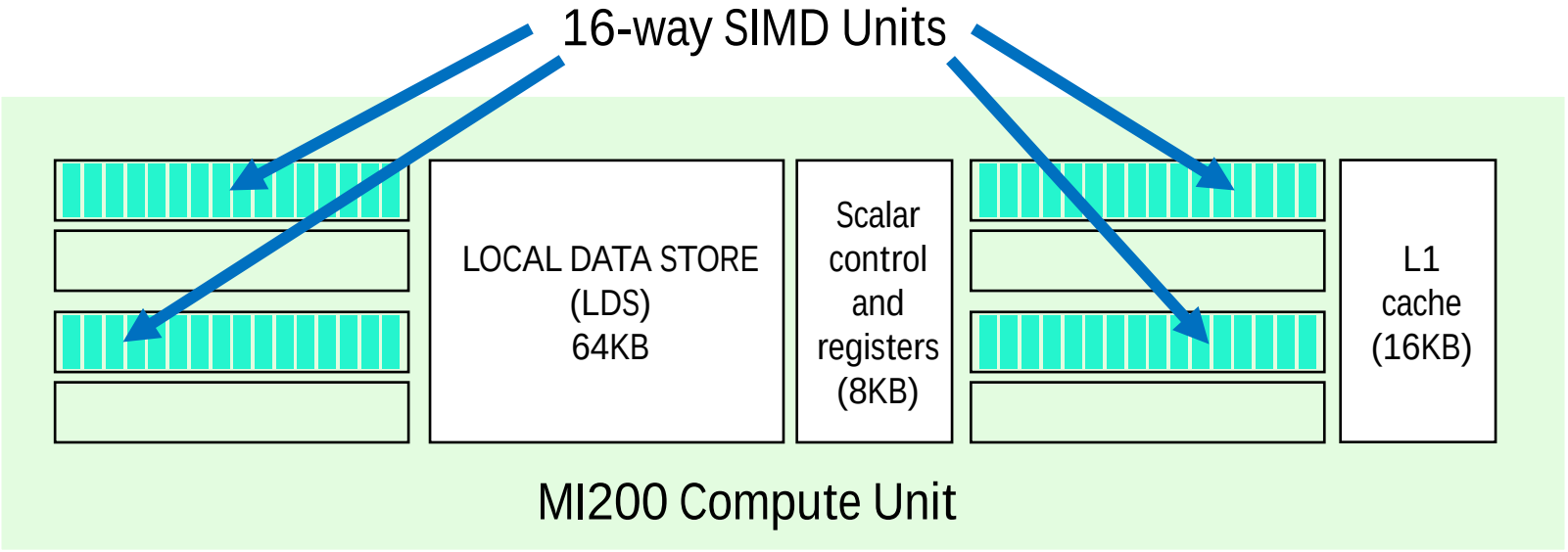


- These are the resources a kernel is launched on



© Copyright 2024 Hewlett Packard Enterprise Development LP

Consider Real Hardware(2): AMD MI-200 Compute Unit (CU)



- Work-groups (blocks) can share the Local Data Store memory
- Each SIMD unit operates on 64 work items (called a wavefront) at once and taking 4 cycles
- Each SIMD unit can handle 10 wavefronts simultaneously
- The maximum number of work-items per workgroup is 1024 (16 wavefronts)
- General advice is to have at least 4 wavefronts in a block (256 work-items)
- Then we need enough blocks to keep the GPU busy (or occupy it)

© Copyright 2024 Hewlett Packard Enterprise Development LP

Now we can define a kernel to scale our vector v?

- We need to mark a function with the qualifier `__global__` to make it a 'kernel' that can be launched from the host…

```
__global__ void scale_vector(int n, double *d_v, double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
     d_v[ i ] *= scale;
   }
  }
```

Compare host code:

for(i=0;i<n;i++){v[i]*=scale;}

- Note that we receive the arguments from the host (more later)
- Pointer arguments need to be valid on the device
- The function is executed for every work item (or thread)
  Many, if not all work items may be executing simultaneously
- Here we are defining a global work item index (i) to span the vector checking we are in bounds

© Copyright 2024 Hewlett Packard Enterprise Development LP

More about kernels

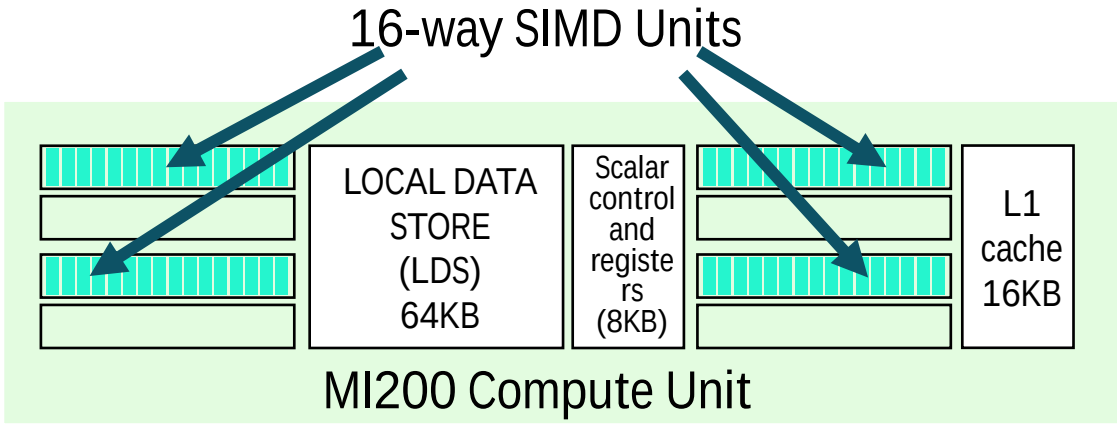
```
__global__ void scale_vector(int n, double *d_v, double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
     v[ i ] *= scale;
   }
  }
```

- We need to take care that threads access independent data,
  we cannot write into or read from a location written by another thread without special techniques
- Private variables (including arrays) are stored locally in each thread (work item)
- A range of mathematical functions are supported as defined in the ROCm documentation:
  https://rocm.docs.amd.com/projects/HIP/en/docs-5.7.0/reference/kernel_language.html
  eg `sin() exp() cos()` etc. and float equivalents

© Copyright 2024 Hewlett Packard Enterprise Development LP

More about kernels(2)

```
__global__ void scale_vector(int n, double *d_v,
                             double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
      if (even(i)) v[ i ] *= scale;
   }
  }
```



- Remember a wavefront at a time is running in SIMD units
- These are running in lock-step executing the same instructions
- This programming model has been called SIMT (Single-Instruction Multiple Threads)
- This means our wavefronts should do the same operations otherwise each step will only progress on some work-items
- This aspect of GPUs is called divergence (the case when the computation diverges between loop items)
- It is OK for wavefronts to diverge from each other

© Copyright 2024 Hewlett Packard Enterprise Development LP

Function Qualifiers

`__global__`

- For functions implementing ‘kernels’ that execute on the device and can only be launched from the host
- Note that kernels are special and need to be launched with appropriate execution context either using chevron syntax or via hipLaunchKernelGGL(...)

`__device__`

- For functions that execute on a device and may only be called from device code:

`__host__`

- For functions that execute on the host and are called from host code

`__host__ __device_`

- For functions that either are called and executed on host or called and executed on the device

© Copyright 2024 Hewlett Packard Enterprise Development LP

Launching the kernel

- In the host code we can launch the kernel as follows…
- We first define the dimensions of the grid and each block

```
const int threads_per_blk = 256;
const int blocks_in_grid = ceil( float(nmax) / thr_per_blk );
dim3 block(threads_per_blk);
dim3 grid(blocks_in_grid);
```

- Note that we don't need to supply all 3 arguments to dim3.
- Now to launch the kernel

```
scale_vector <<< grid, block >>> (N, d_v, 1000.0);
```

- An alternative to <<< >>> syntax is to use the hipLaunchKernelGGL(…) function.

© Copyright 2024 Hewlett Packard Enterprise Development LP

Complete code

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

#define HIP_CHECK(hip_api_call) {                    \
  hipError_t herr = hip_api_call; \
  if( herr ) { std::cerr << "Error(HIP): " << hipGetErrorString(herr) << endl;
std::abort(); } }


__global__ void scale_vector(int n, double *v, double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
     v[ i ] *= scale;
   }
  }
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

Complete code...

```
int main(int argc, char** argv){

  const int N=4096;
  int i;
  double *v = (double *) malloc( (size_t) N * sizeof(double));
  double *d_v = NULL;

  size_t v_nbytes = N * sizeof(double);
  hipMalloc(&d_v, v_nbytes);

  for(i=0;i<N;i++){ v[i] = i;}
  hipMemcpy(d_v, v, v_nbytes, hipMemcpyHostToDevice);

  const int thr_per_blk = 256;
  const int blks_in_grid = ceil( float(N) / thr_per_blk );

  scale_vector <<< blks_in_grid, thr_per_blk >>> (N, d_v, 1000.0);

  hipMemcpy(v, d_v, v_nbytes, hipMemcpyDeviceToHost);

  for(i=0;i<9;i++){printf("v[i]=%.0f ",v[i]);printf("\n");}
  free(v);
  hipFree(d_v);
  return EXIT_SUCCESS;
}
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

Build and execute code

- Compile with programming environment (CCE or AMD compilers) or ROCm hipcc

  cc –x hip –o vector vector.cpp

  hipcc --offload-arch=gfx90a -o vector vector.cpp

- Run…

```
> srun … ./vector
srun: job 7969957 queued and waiting for resources
srun: job 7969957 has been allocated resources
v[i]=0
v[i]=1000
v[i]=2000
v[i]=3000
v[i]=4000
v[i]=5000
v[i]=6000
v[i]=7000
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

CUDA to HIP

In most cases you can just replace ‘cuda’ with ‘hip’ in the API

- Instead of

```
#include <cuda.h>
```

- use

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

- CUDA and HIP use different compilers provided by the CUDA toolkit or ROCm
  - nvcc for NVIDIA (note that two compilation phases are used, one for host and one for device)
  - hipcc open-source clang-based compilers when used for AMD/HIP, uses nvcc for CUDA backend.

© Copyright 2024 Hewlett Packard Enterprise Development LP

Previous Example: CUDA and HIP comparison

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>

#define CUDA_CHECK(cuda_api_call) { cudaError_t herr = cuda_api_call; \
  if( herr ) { std::cerr << "Error(CUDA): " << cudaGetErrorString(herr) <<
endl; std::abort(); } }
__global__ void scale_vector(int n, double *v, double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
     v[ i ] *= scale;
   }  }
int main(int argc, char** argv){
  const int N=4096;
  int i;
  double *v = (double *) malloc( (size_t) N * sizeof(double));
  double *d_v = NULL;

  size_t v_nbytes = N * sizeof(double);
  cudaMalloc(&d_v, v_nbytes);

  for(i=0;i<N;i++){ v[i] = i;}
  cudaMemcpy(d_v, v, v_nbytes, cudaMemcpyHostToDevice);
  const int thr_per_blk = 256;
  const int blks_in_grid = ceil( float(N) / thr_per_blk );

  scale_vector <<< blks_in_grid, thr_per_blk, 0, 0 >>> (N, d_v, 1000.0);

  cudaMemcpy(v, d_v, v_nbytes, cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  for(i=0;i<9;i++){printf("v[i]=%.0f ",v[i]);printf("\n");}
  free(v);
  cudaFree(d_v);
  return EXIT_SUCCESS; }
```

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

#define HIP_CHECK(hip_api_call) { hipError_t herr = hip_api_call; \
  if( herr ) { std::cerr << "Error(HIP): " << hipGetErrorString(herr) <<
endl; std::abort(); } }
__global__ void scale_vector(int n, double *v, double scale ){
  int i = threadIdx.x + blockIdx.x*blockDim.x ;
   if ( i < n) {
     v[ i ] *= scale;
   } }
int main(int argc, char** argv){
  const int N=4096;
  int i;
  double *v = (double *) malloc( (size_t) N * sizeof(double));
  double *d_v = NULL;

  size_t v_nbytes = N * sizeof(double);
  hipMalloc(&d_v, v_nbytes);

  for(i=0;i<N;i++){ v[i] = i;}
  hipMemcpy(d_v, v, v_nbytes, hipMemcpyHostToDevice);
  const int thr_per_blk = 256;
  const int blks_in_grid = ceil( float(N) / thr_per_blk );

  scale_vector <<< blks_in_grid, thr_per_blk, 0, 0 >>> (N, d_v, 1000.0);

  hipMemcpy(v, d_v, v_nbytes, hipMemcpyDeviceToHost);
  hipDeviceSynchronize();

  for(i=0;i<9;i++){printf("v[i]=%.0f ",v[i]);printf("\n");}
  free(v);
  hipFree(d_v);
  return EXIT_SUCCESS; }
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

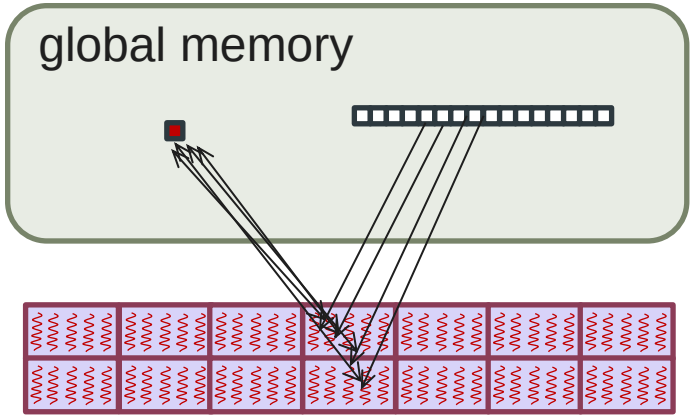
Reminder on CUDA vs HIP terminology

AMD	NVIDIA/CUDA	Description
Compute Unit	Streaming Multiprocessor	Parallel vector processor
Wavefront	Warp	NVIDIA warp size is 32 threads AMD wavefront size is 64 work items
Work group/ block	Thread block	Group of work items/threads on a Compute Unit / Streaming Multiprocessor of the GPU at the same time
Work item	Thread	Individual item of work
Local Memory	Shared Memory	Shared between wavefronts in a workgroup
Private Memory	Local Memory	Per-thread private memory, often mapped to registers
Global Memory	Global Memory	RAM globally accessible via cache hierarchy

© Copyright 2024 Hewlett Packard Enterprise Development LP

Data races and atomics

- HIP provides access to the same memory locations from multiple work-items
- This opens us up to the danger of data races which happen when
  - Two or more threads access the same location (and at least one is a write)
  - Although accesses are programmatically simultaneous there is no defined ordering
- The danger is from the fact that work-items may be scheduled at different times
- Typical scenarios when this might happen:
  - We sum values into a shared location
  - We look for a minimum or maximum value continuously updating a shared location



© Copyright 2024 Hewlett Packard Enterprise Development LP

Thread synchronisation and atomics

- One way around the problem with updating a shared location is to use atomic operations
- HIP provides many atomics, some examples are:

  atomicAdd(int *address, int val) (also long, float and double arguments)
  atomicSub(int *address, int val) ...
  atomicMax(int *address, int val) ...
  atomicMin(int *address, int val) ...

  along with Exchange, compare and swap etc.
- An atomic operation guarantee that no other thread can read or modify the target during the operation.
- Example:

```
__global__ void max_vector(int n, double *vmax, double *v ){
 int i = threadIdx.x + blockIdx.x*blockDim.x ;
   atomicMax(vmax,v[i]);
}
```

- Note that hardware and API support for atomic operations varies by ecosystem and platform.

© Copyright 2024 Hewlett Packard Enterprise Development LP

Hierarchical reduction approach

- You may recall that a CU has a Local Data Store an area of share memory accessible to a workgroup/block
- We can allocate memory into the LDS with the __shared__ type qualifier in kernel code, e.g..

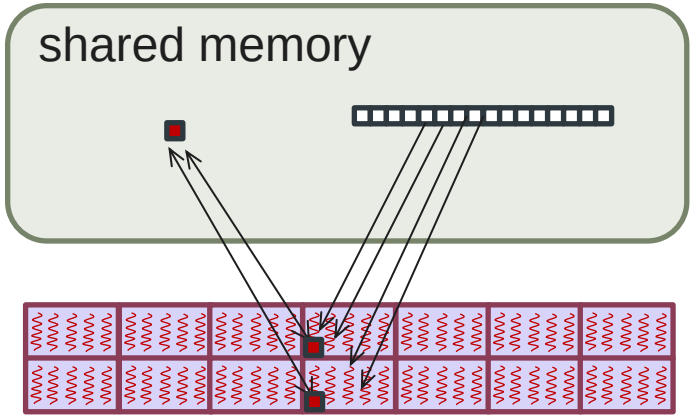
  __shared__ double lds_vmax;

- An alternative way to do our max calculation is to perform it into a variable in LDS and then into global memory

- We can synchronise threads in a workgroup, they only progress after all have reached the syncthreads().

  __syncthreads()

- This would be crucial to implement this method (exercise)



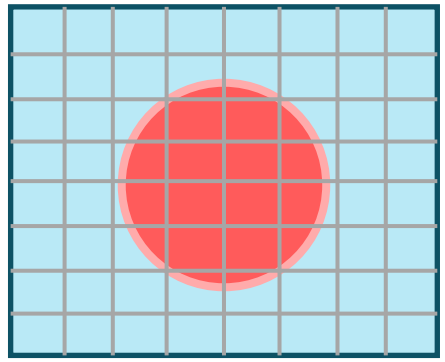
Algorithm:

- All threads in each block update variable in LDS
- Threads synchronise
- One thread in block updates variable in global memory

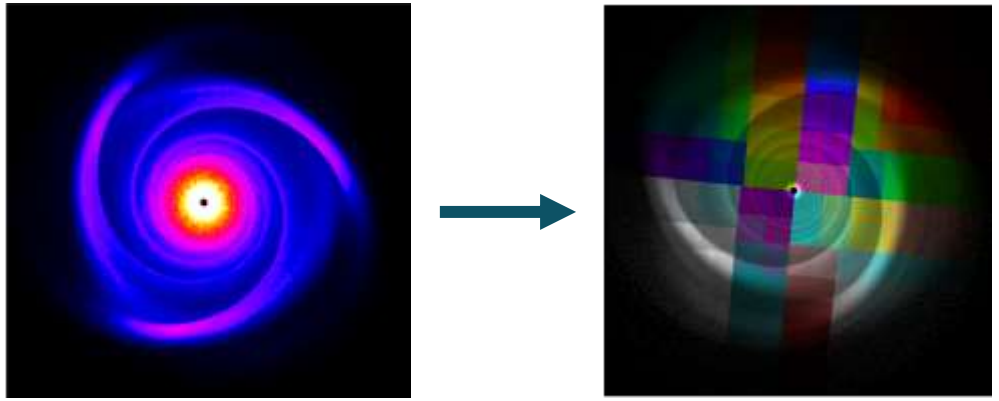
© Copyright 2024 Hewlett Packard Enterprise Development LP

Mapping our problem to the Grid

- HIP and CUDA can naturally be used to map problems on a domain to the Grid.
- If you don't have such a problem then you can still map it to the 1-D grid
  - For example an n-body simulation with a list of particles
- You can also just use a 1-D grid and map those coordinates into higher dimension data structures as needed
  (eg. `a[i][j]` in 2D maps to `a[i*stride + j]` in 1D)
- A kernel may access selected dimensions of multidimensional data by looping locally
  - So consider decomposing a 2D problem
  - Parallelise first dimension of data across threads
  - Each thread loops over other dimension



2d domain decomposition



Planetary formation:
particles moved to 1d grid

© Copyright 2024 Hewlett Packard Enterprise Development LP

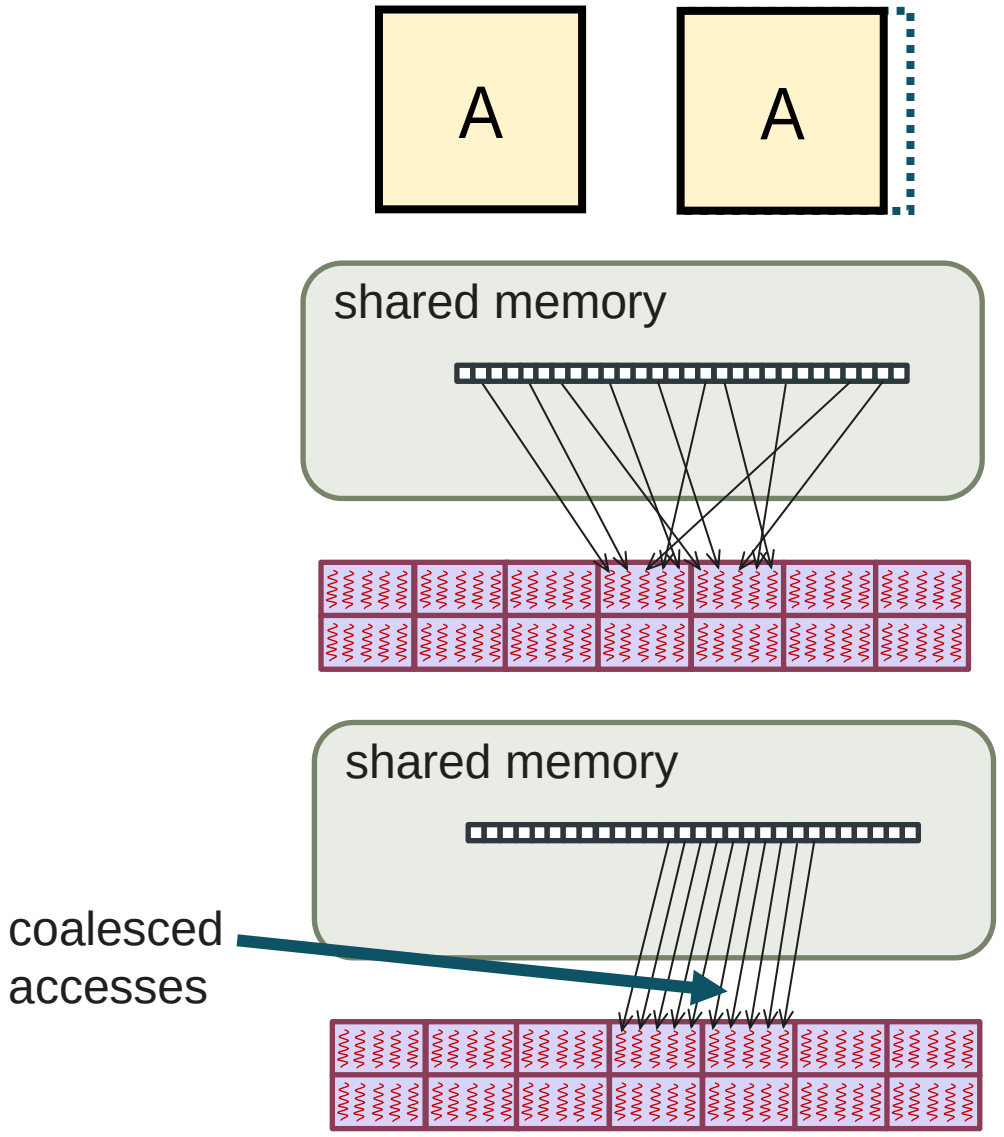
Memory access

A typical scenario is to access a 2D array in global memory

- It will be more efficient if the array and workgroup dimensions are multiple of the wavefront size

For accesses via L2 and for local memory:

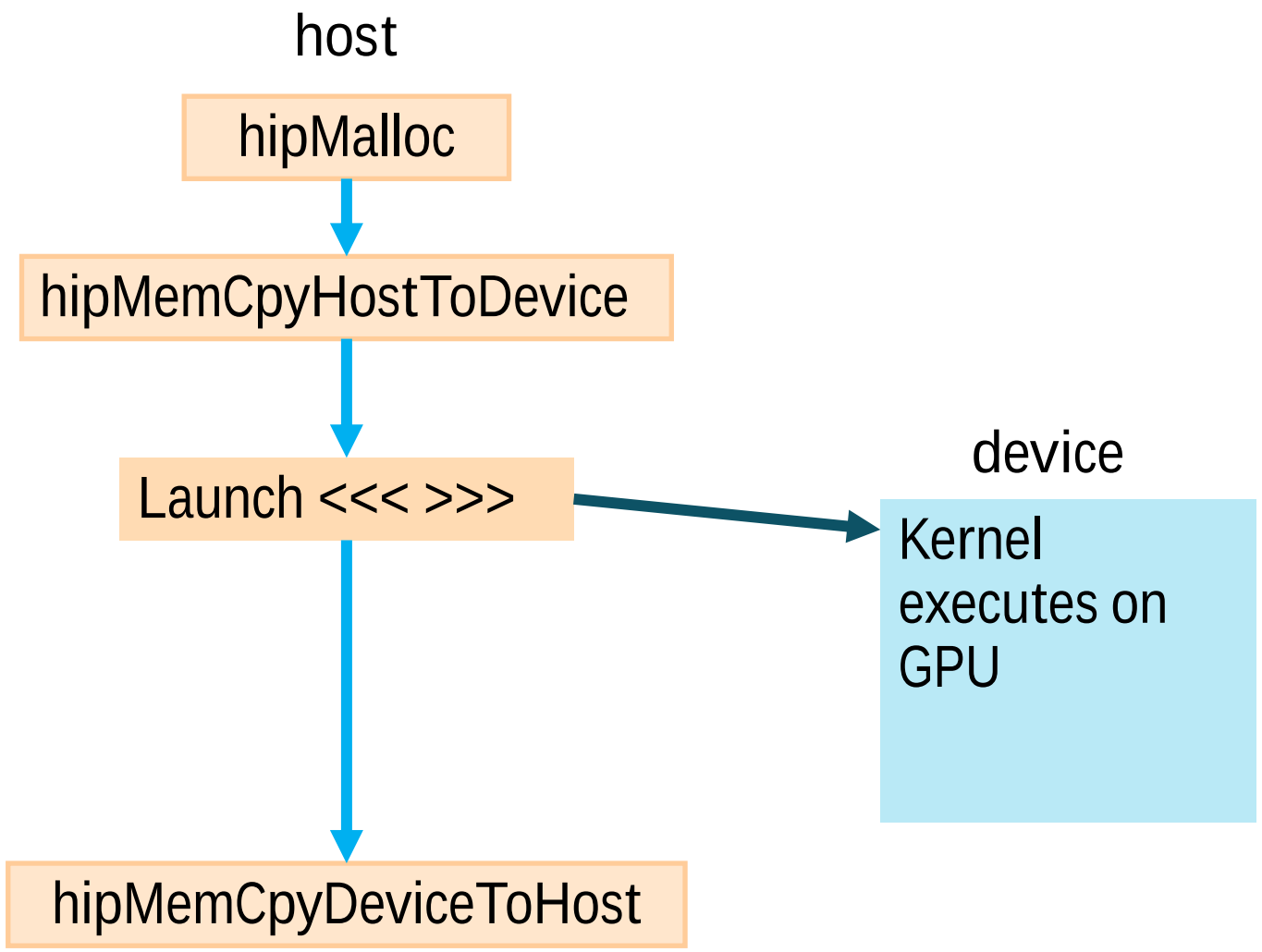
- It is more efficient if the threads in a warp access consecutive local memory locations
- This is termed coalesced memory access



© Copyright 2024 Hewlett Packard Enterprise Development LP

Synchronisation

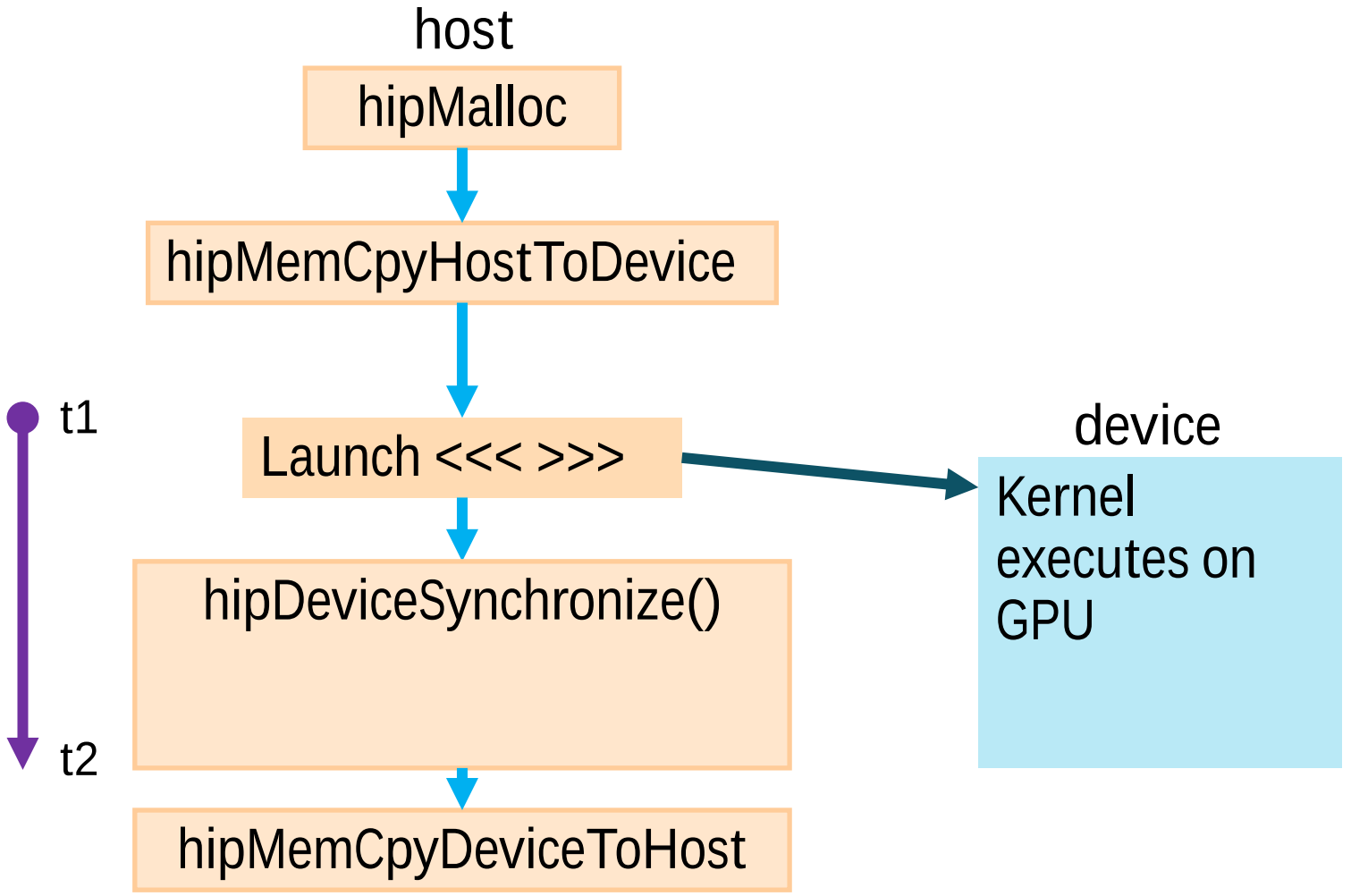
- Some operations like kernel launch are asynchronous
- We can't be sure when the kernel completes
- The runtime progresses GPU operations in-order in a stream
- Hence, we know the kernel has completed after the device to host copy.



© Copyright 2024 Hewlett Packard Enterprise Development LP

Be aware of asynchronous execution when timing a kernel…

- You can't just call a timer after the kernel launch
- We can use hipDeviceSynchronize() to be sure operations are completed



© Copyright 2024 Hewlett Packard Enterprise Development LP

Recap

- Simple generic CPU/GPU architecture
- The basics of programming in HIP
- First steps, use the HIP API to detect GPUs
- Fundamental parts of a HIP application
  - Memory allocation
  - Copying data to the GPU
  - Executing a workload on the GPU
  - Copying data from the GPU
- Mapping workloads to the GPU and how hardware comes into this
- API comparison with CUDA
- Other considerations

© Copyright 2024 Hewlett Packard Enterprise Development LP

More Advanced Features of HIP

© Copyright 2024 Hewlett Packard Enterprise Development LP

Streams and Asynchronous HIP usage

- Recall that by default a set of memory operations and kernel launches will progress in sequence on a default steam (this is called the NULL stream)…

```
hipMemcpy(…)
hipMemcpy(…)
kernel1 <<< dim3(1), dim3(1024) >>> (N,d_v1)
kernel2 <<< dim3(1), dim3(1024) >>> (N,d_v2)
hipMemcpy(…)
hipMemcpy(…)
```

- In this case the kernels are using minimal GPU resources so the GPU is mostly idle
- We have to wait until the memory copies complete and cannot do any work on the cpu while that happens

© Copyright 2024 Hewlett Packard Enterprise Development LP

Asynchronous Memory Copy

- There is an asynchronous memory-copy API call
  hipMemcpyAsync(d_a , h_a , Nbytes , hipMemcpyHostToDevice , stream)
  which does not block.
  (Note that we need to allocate page-locked/pinned memory with cudaHostMalloc() when using this.)
- It operates on a stream which we can initialize and destroy with

```
hipStream_t steam;
hipSteamCreate(&stream);
...
hipSteamDestroy(stream);
```

- Tasks queued in a stream progress in-order but tasks in different streams may share resources and overlap.
- Any task on the NULL stream (used by passing 0 as a stream argument) will only progress when all tasks on other streams complete
- We can use hipStreamSynchronize(strean) to block until all work on the stream completes.

© Copyright 2024 Hewlett Packard Enterprise Development LP

Now we can be as asynchronous as possible…

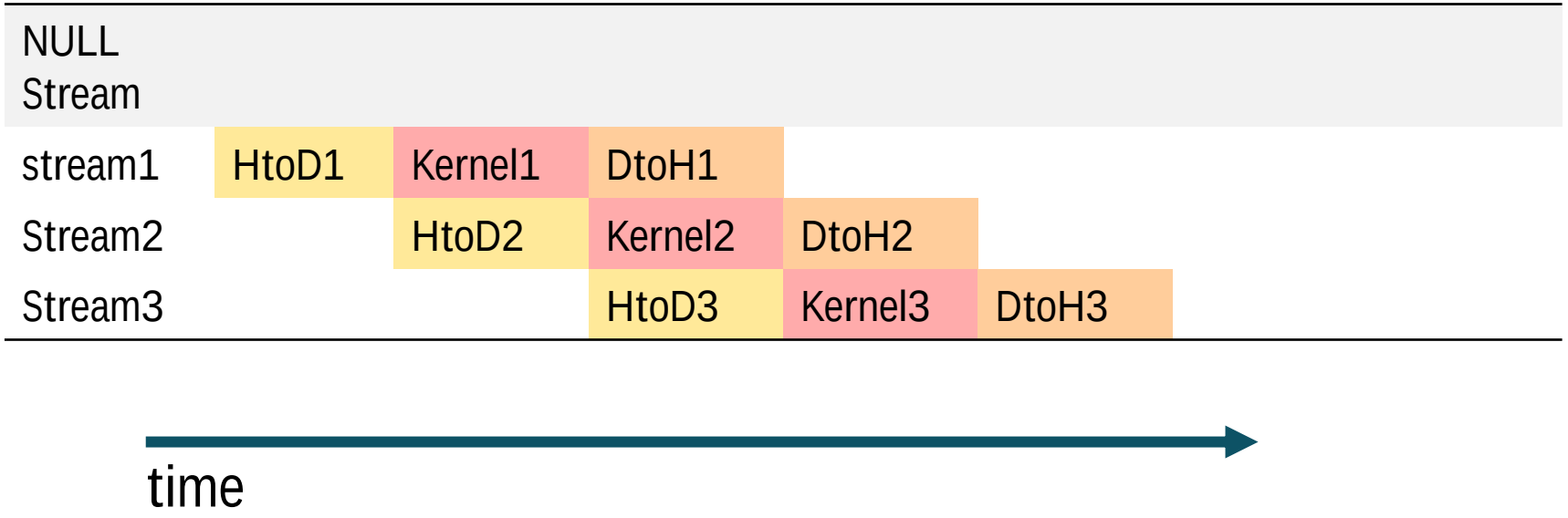
```
hipMemcpyAsync(d_v1, h_v1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_v2, h_v2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_v3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);
kernel1 <<< dim3(1), dim3(1024), 0, stream1 >>> (N,d_v1);
kernel2 <<< dim3(1), dim3(1024), 0 ,stream2 >>> (N,d_v2);
kernel3 <<< dim3(1), dim3(1024), 0, stream3 >>> (N,d_v3);
hipMemcpyAsync(h_v1, d_v1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_v2, d_v2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_v3, d_v3, Nbytes, hipMemcpyDeviceToHost, stream3);
```

- Note that we are supplying a stream to the kernel launches.
- The GPU can progress memory copies from host to device, memory copies from device to host and kernels at the same time

© Copyright 2024 Hewlett Packard Enterprise Development LP

Stream execution

- This is what we might see...



© Copyright 2024 Hewlett Packard Enterprise Development LP

So when would these features be useful

- Allowing the host to progress work on the cpu at the same time as memory copies to/from the device
- Help to keep the GPU busy by launching multiple kernels if each does not fill the GPU.
- This can be important when data has to be moved to other processes, for example an MPI application.

© Copyright 2024 Hewlett Packard Enterprise Development LP

More on shared/local memory

- Recall that shared memory may be allocated in a kernel to be used as a temporary scratch area:

```
__global__ transform(const int M, const int N, const double *A,
                     double* Atrasformed) {
__shared__ double s_A[512];
```

- It is possible to launch a kernel and to provide the amount of shared memory to be used for one allocation in the kernel:

```
__global__ transform(const int M, const int N, const double *A,
                     double* Atrasformed) {
HIP_DYNAMIC_SHARED(double, s_A);
```

- Called from

```
size_t Nsharedbytes=M*N*sizeof(double);
transform <<< grid, block, Nsharedbytes, 0 >>>(M,N,d_A,d_Atransformed)
```

© Copyright 2024 Hewlett Packard Enterprise Development LP

Using Libraries

- Both NVIDIA and AMD have an ecosystem of scientific libraries that provide optimized implementation of various algebra. For example, Fourier transforms, sparse and dense linear algebra etc.
- Check if any libraries exist that are relevant for your application.

- An example of using a dense linear algebra library…
- Perform the general matrix multiply operation C = alpha*op(A)*op(B) + beta*C where A,B,C are arrays which may be transposed.
- The BLAS routine for this is sgemm.
- We can call the hip routine as follows but we need to define the required parameters…

```
hipblasSgemm(handle, transa, transb,
             m, n, k, &alpha, da, lda, db, ldb, &beta, dc, ldc));
```

All we had to do was define the relevant arguments (array sizes and embedding and operators) and pass in the pointers to gpu data

© Copyright 2024 Hewlett Packard Enterprise Development LP

Using Multiple GPUs

- It is very typical for applications to use more than one GPU. In HPC contexts this tends to mean that people are running one MPI process per GPU.
- We can use more than one GPU in HIP by using the device API to select a GPU to subsequently use.

```
hipSelectDevice(i);
```

- We can check how many devices are visible using

```
hipDeviceCount();
```

- In order to use more than one device effectively it would be required to use streams and asynchronous operations or to use threads (but only address one device from a thread)

© Copyright 2024 Hewlett Packard Enterprise Development LP

GPU to GPU communication

- When using multiple GPUs we need to consider how to communicate data from GPU to GPU and what programming models are used.
- For a developer of an HPC application the most likely use-case is that the programming model is multiple MPI processes each accessing one GPU.
- In this case the MPI implementation may be able to communicate data from GPU to GPU directly and not copy data back to the host (where MPI is running). You should find out if this is supported on any system you are using.
- On Dardel this can be enabled by setting this…

```
export MPICH_GPU_SUPPORT_ENABLED=1
```

- There are lower-level means to move data between GPUs such as RCCL from AMD and NCCL from AMD, these are most used in AI frameworks.
- We will not cover these topics.

© Copyright 2024 Hewlett Packard Enterprise Development LP

High-level approach to optimization for GPU

Consider the following aspects

- The application needs to expose enough parallelism to keep the device busy
- Minimise data transfers between host to device
  - Could batch smaller transfers
  - Using mapped page-locked memory avoids the manual data movement
  - On hardware with integrated device/host memory (MI300A) use that memory.
- Use streams and asynchronous features to help hide data movement and allow for multiple kernels
- Launch multiple-kernels at once if occupancy will be an issue
- Run on host if there is little parallelism in some application areas.
- Avoid control flow in wavefronts/warps (divergent wavefronts/warps)
- Organize global memory accesses to be in a friendly pattern
- Consider staging data from global memory into local data store on the CUs (like a user-managed cache)

© Copyright 2024 Hewlett Packard Enterprise Development LP

References

ROCm and HIP programming

- AMD ROCm documentation (latest version): https://rocm.docs.amd.com/en/latest/
- AMD HIP Programming guide (ROCm 5.2): https://raw.githubusercontent.com/RadeonOpenCompute/ROCm/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf
- There are also very many training slides from AMD and large sites that cover HIP

More hardware oriented.

- AMD CDNA2 Architecture: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf
- "AMD Instinct MI200" Instruction Set Architecture ***Reference Guide,*** https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf

© Copyright 2024 Hewlett Packard Enterprise Development LP