High-Performance Computing

Daniele De Sensi

Info on Exam

Exam Dates

- Jan: January 20th 9:00 AM
- Feb: February 9th 9:00 AM
- Jun: On-demand
- Jul: On-demand
- Sep: On-demand

Exam Content

- 2 non-mandatory (but highly suggested) practical tasks
  - For Prof. Pontarelli part: https://classroom.github.com/a/upZ5q--C
  - For Prof. De Sensi part: https://classroom.github.com/a/B56wB6Bd
    - Adapt the Jacobi code to do 2D domain decomposition (pay attention to columns being non contiguous in memory)
    - You can assume the num rows to be divisible by the number of GPUs per row (2), and the num columns to be divisible by the number of GPUs per column (2).
    - If you want to try with having 2 GPUs per node, since you can allocate at most 4 nodes, you should be able to try on up to 4x2=8 GPUs (arranged as 2x4 or 4x2)
    - You should use either NCCL or NVSHMEM (or both, if you like)
  - Prepare a short report for both parts (analyze scalability, profiling, etc...)
  - Discussion of the report/project
- Traditional oral exam (Q&A)

Practical Task – 1° part

Assignment: correlated pairs

You are given m input vectors, each with **n** numbers. Your task is to calculate the correlation between every pair of input vectors. You need to implement the following function: *void correlate(int ny, int nx, const float* data, float* result)* Here *data* is a pointer to the input matrix, with **ny** rows and **nx** columns. For all 0 <= y < ny and 0 <= x < nx, the element at row *y* and column *x* is stored in *data[x + y*nx]*. The function must solve the following task: for all i and j with 0 <= j <= i < ny, calculate the correlation coefficient between row i of the input matrix and row j of the input matrix, and store the result in *result[i + j*ny]*.

Correlation coefficient

Given two rows with elements {x1..xn} and {y1..yn} the correlation(1) rxy is defined as:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where:

- **n** is the number of samples
- $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the sample mean

Practical Task – 1° part

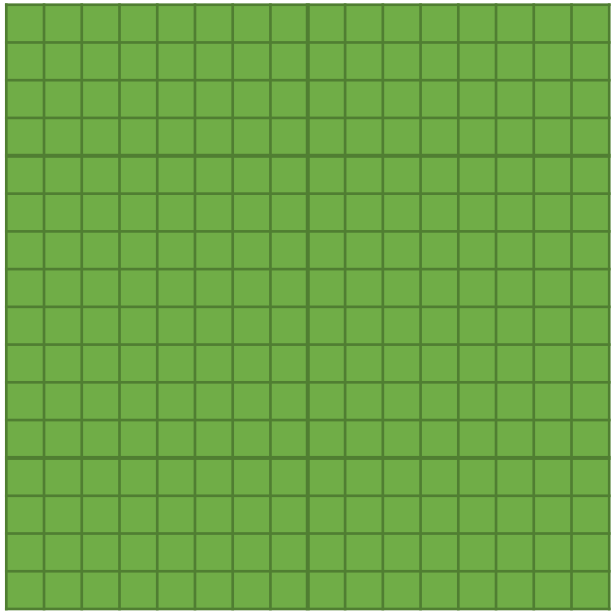
Assignment:

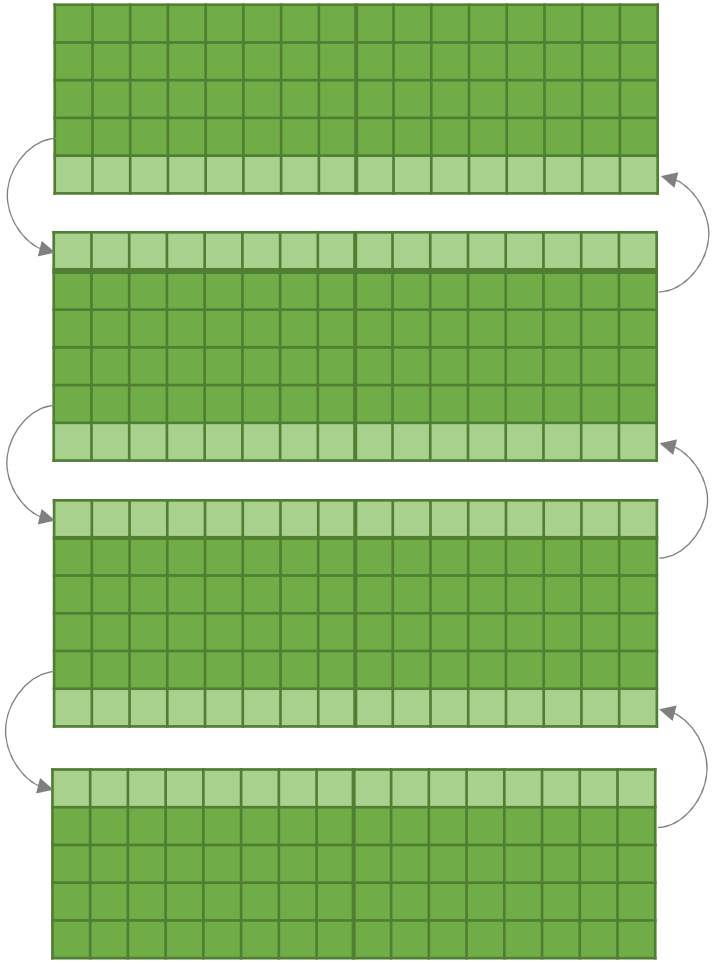
For the correlated pair function:

- Implement a simple baseline solution
- Using only one CPU core, solve the task as fast as possible.
- Measure the arithmetic intensity (AI) and the memory BW of the baseline solution and of the fast solution
- Plot the results on a roofline graph
- Measure the cache miss rate for the baseline solution and of the fast solution
- Identify the source code line on which the baseline and fast solutions spend more time
- Identify the source code line on which the baseline and fast solutions have more LLC cache misses

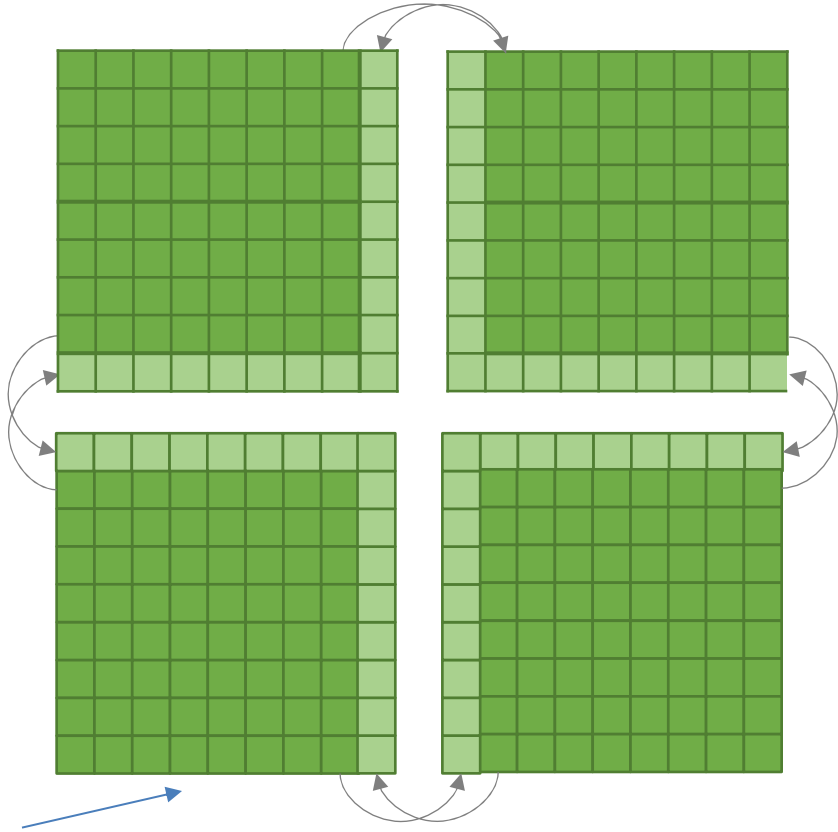
Write a report explaining the collected data.

Practical Task - 2° part





* Minimize number of neighbors: Communicate to fewer neighbors
* Optimal for latency bound communication,
* Contiguous Transfers



* Minimize surface area/volume ratio: Communicate less data
* Optimal for bandwidth bound communication
* Non-Contiguous Transfers

Recap

NCCL-API (With MPI) - Initialization

First, we need a NCCL-Communicator for, this, wee need a NCCL UID

We use MPI-Ranks and size for initialization

```
MPI_Init(&argc,&argv)
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

ncclUniqueId nccl_uid;
if (rank == 0) ncclGetUniqueId(&nccl_uid);
MPI_Bcast(&nccl_uid, sizeof(ncclUniqueId), MPI_BYTE, 0, MPI_COMM_WORLD));

ncclComm_t nccl_comm;
ncclCommInitRank(&nccl_comm, size, nccl_uid, rank);
…
…
ncclCommDestroy(nccl_comm);
MPI_Finalize();
```

Fused Communication Calls

- Multiple calls to ncclSend() and ncclRecv()
  should be fused with ncclGroupStart() and ncclGroupEnd()to
  - Avoid deadlocks
    (if calls need to progress concurrently)
  - Reduces communication kernels launch overhead
  - Communication effectively start when ncclGroupEnd() is called
    - i.e., it guarantees it has been enqueued to the stream, not that it completed

SendRecv:

```
ncclGroupStart();
 ncclSend(sendbuff, sendcount, sendtype, peer, comm, stream);
ncclRecv(recvbuff, recvcount, recvtype, peer, comm, stream);
ncclGroupEnd();
```

Bcast:

```
ncclGroupStart();
if (rank == root) {
  for (int r=0; r<nranks; r++)
    ncclSend(sendbuff[r], size, type, r, comm, stream);}
ncclRecv(recvbuff, size, type, root, comm, stream);
ncclGroupEnd();
```

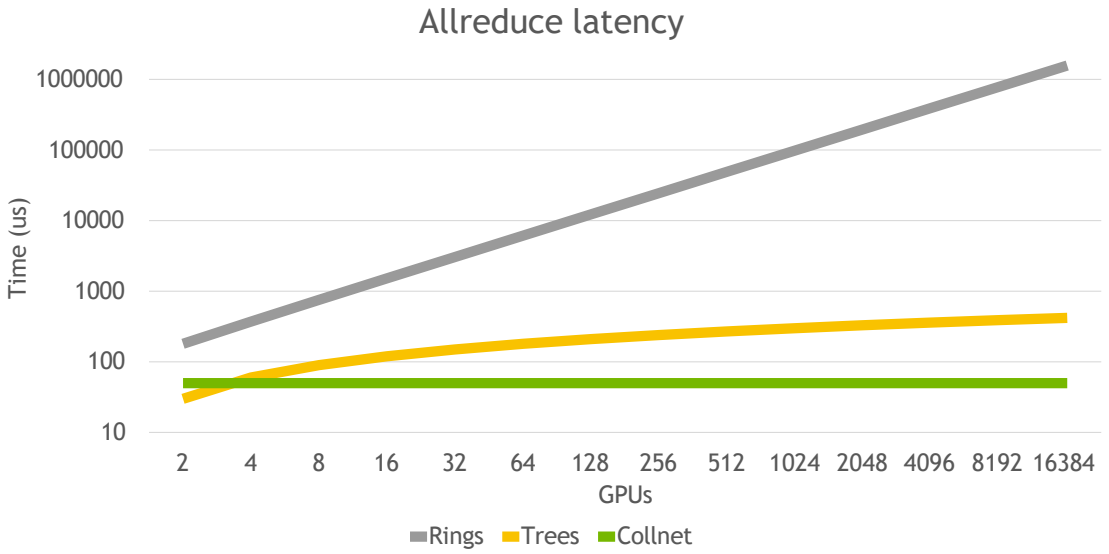
Neighbor exchange:

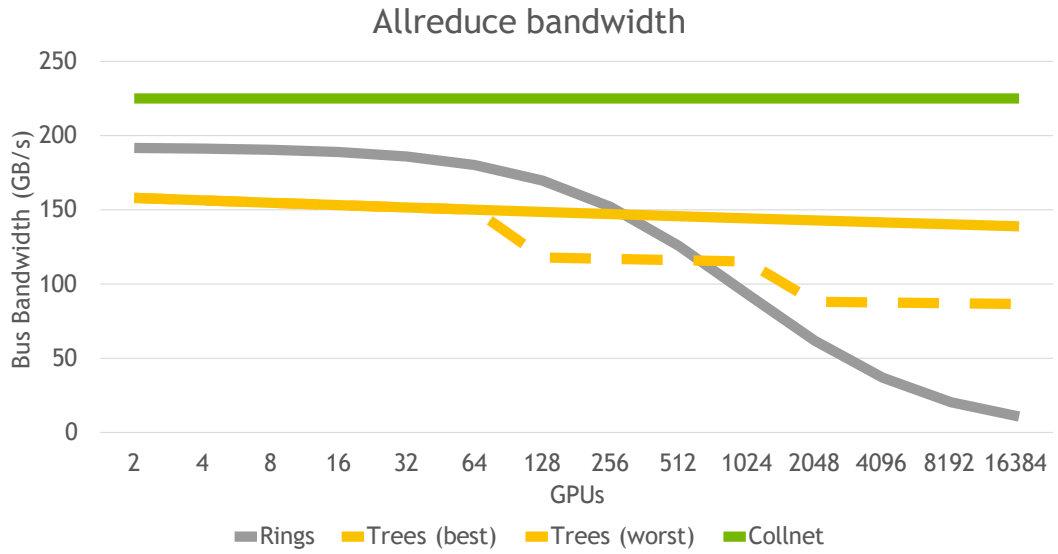
```
ncclGroupStart();
for (int d=0; d<ndims; d++) {
  ncclSend(sendbuff[d], sendcount, sendtype, next[d], comm, stream);
  ncclRecv(recvbuff[d], recvcount, recvtype, prev[d], comm, stream);}
ncclGroupEnd();
```

Algorithms Summary

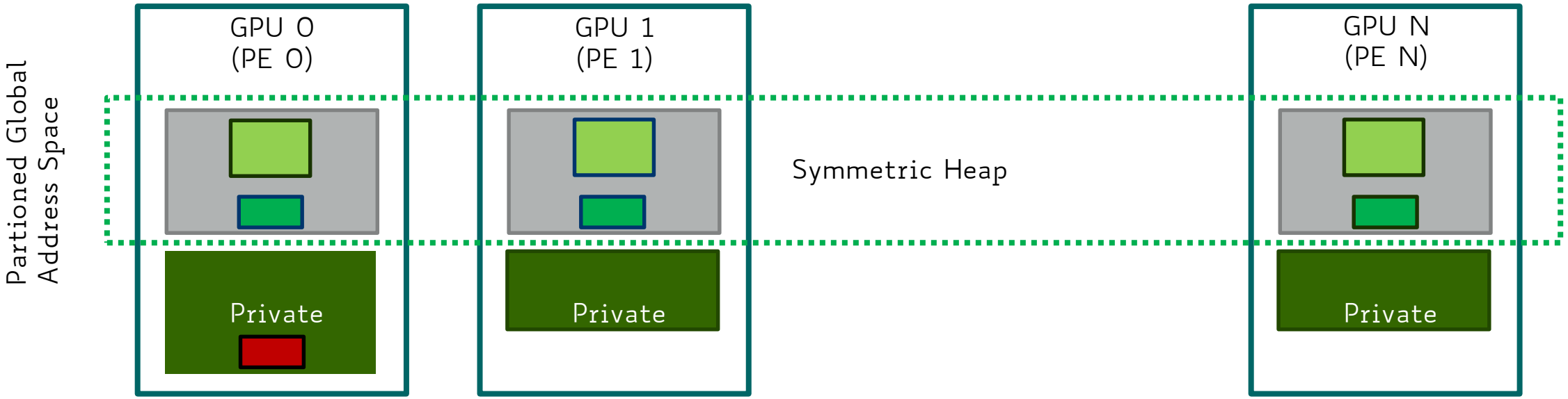
Pros and cons

Algorithm	Latency	Bandwidth	Computing	Network Pattern
Rings	Linear	Perfect	Uniform	Few flows
Trees	Log	Close to perfect	Imbalanced	Many flows
Collnet	Constant	Close to 2x (may be limited by NVLink)	Almost uniform	Minimal flows





NVSHMEM Symmetric Memory Model

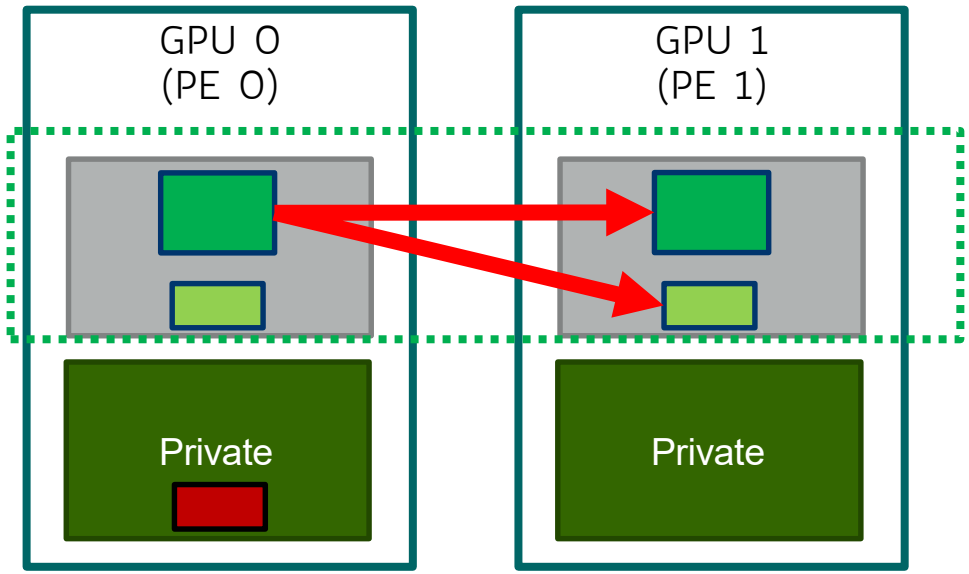


Symmetric objects are allocated collectively with the same size on every PESymmetric memory: nvshmem_malloc(shared_size);

Must be the same on all PEs

Private memory:
cudaMalloc(...)

NVSHMEM Host API Put



copies *nelems* data elements of type T from symmetric objects *src* to *dest* on PE *pe*

```
void nvshmem_<T>_put                (T* dest, const T* src, size_t nelems, int pe);
void nvshmemx_<T>_put_on_stream(T* dest, const T* src, size_t nelems, int pe,
                                    cudaStream_t stream);
```

The x marks extensions to the OpenSHMEM API

ATTENTION: Are asynchronous calls, when they return, I do not know if data was written in remote memory already

CUDA Graphs

From streams and events to task graphs

- Limitations of scheduling task graphs with streams
  - Sub-optimal scheduling : GPU runtime has no vision of tasks ahead
  - Must pay various initialization overheads when launching each task

- New alternative since CUDA 10.0: cudaGraph API
  - Build a representation of the dependency graph offline
  - Let the CUDA runtime optimize and schedule the task graph
  - Launch the optimized graph as needed

- Two ways we can build the dependency graph
  - Record a sequence of asynchronous CUDA calls
  - Describe the graph explicitly

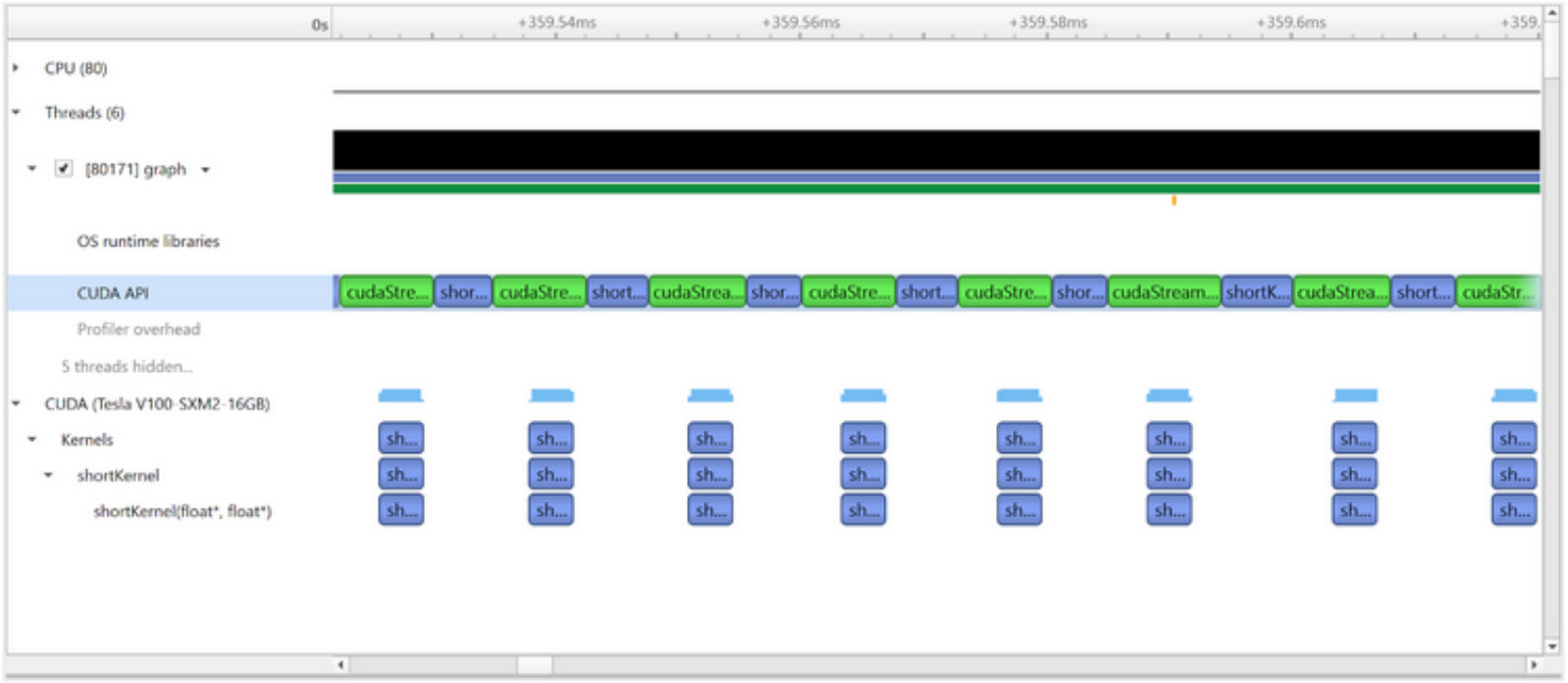
Kernel overhead launch

```
#define N 500000 // tuned such that kernel takes a 2.9 microseconds on an Tesla V100 GPU

__global__ void shortKernel(float * out_d, float * in_d){
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if(idx<N) out_d[idx]=1.23*in_d[idx];
}

#define NSTEP 1000
#define NKERNEL 20

// start CPU wallclock timer
for(int istep=0; istep<NSTEP; istep++){
 for(int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
   shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
   cudaStreamSynchronize(stream);
 }
} //end CPU wallclock time
```

The code calls the kernel 20 times, each of 1,000 iterations.

CPU-based wallclock measures 9.6μs per kernel: much higher that the kernel execution time of 2.9μs.

Kernel overhead launch



Total time is the sum of the kernel execution times plus any overheads. We can see this visually using the Nsight Systems profiler

Overlapping Kernel Launch and Execution

```
#define N 500000 // tuned such that kernel takes a 2.9 microseconds on an Tesla V100 GPU

__global__ void shortKernel(float * out_d, float * in_d){
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if(idx<N) out_d[idx]=1.23*in_d[idx];
}

#define NSTEP 1000
#define NKERNEL 20

// start CPU wallclock timer
for(int istep=0; istep<NSTEP; istep++){
 for(int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
   shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
 }
cudaStreamSynchronize(stream);
} //end CPU wallclock time
```

The code calls the kernel 20 times, each of 1,000 iterations.

CPU-based wallclock measures 3.8μs per kernel: still higher that the kernel execution time of 2.9μs.

Overlapping Kernel Launch and Execution

We have removed the synchronization API calls, except the one at the end of the timestep. Launchs overlap with kernel executions, but we are still performing a separate launch operation for each kernel.

CUDA Graph

- We can further improve performance by using a CUDA Graph to launch all the kernels within each iteration in a single operation.

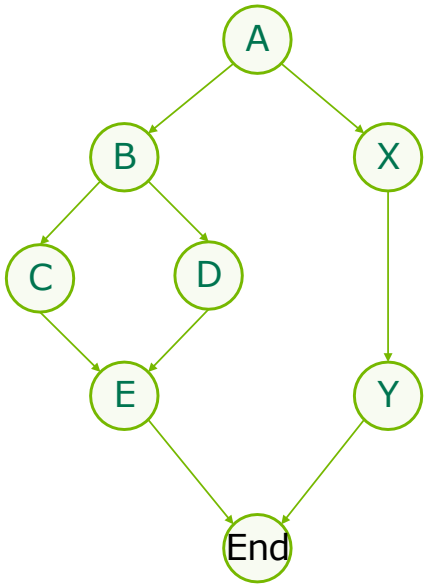
- A graph is a series of operations, such as kernel launches, connected by dependencies, which is defined separately from its execution. This allows a graph to be defined once and then launched repeatedly.

- Work submission using graphs is separated into three distinct stages: definition, instantiation, and execution.
  - During the **definition** phase, a program creates a description of the operations in the graph along with the dependencies between them.
  - **Instantiation** takes a snapshot of the graph template, validates it, and performs much of the setup and initialization of work with the aim of minimizing what needs to be done at launch. The resulting instance is known as an executable graph.
  - An executable graph may be **launched** into a stream, similar to any other CUDA work. It may be launched any number of times without repeating the instantiation.

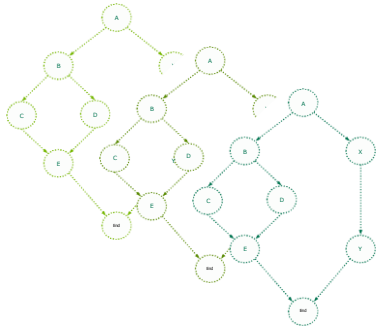
Three-Stage Execution Model

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

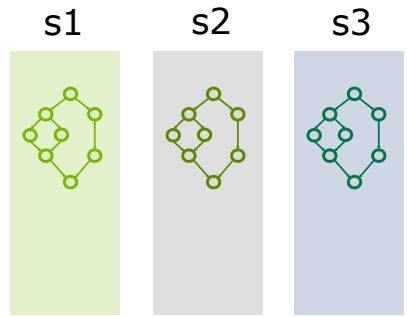
Define



Instantiate



Execute



Single Graph "Template"

Created in host code or built up from libraries

Multiple "Executable Graphs"

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

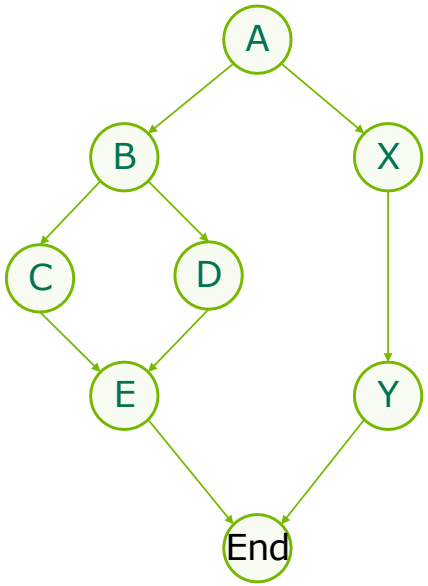
CUDA Graph Node Types

- A graph node can be one of:
  - Kernel
  - CPU function call: *cudaLaunchHostFunc()*
  - memory copy
  - Memset
  - empty node
  - waiting on an event
  - recording an event
  - conditional node
  - child graph: To execute a separate nested graph

Three-Stage Execution Model

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

Define



Single Graph "Template"

Created in host code or built up from libraries

Instantiate

Multiple "Executable Graphs"

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

Execute

Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

Creating a Graph Using Capture

We can further improve performance by using a CUDA Graph to launch all the kernels within each iteration in a single operation.

```
bool graphCreated=false;
cudaGraph_t graph;
cudaGraphExec_t instance;
for(int istep=0; istep<NSTEP; istep++){
   if(!graphCreated){
     cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
     for(int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
       shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
     }
     cudaStreamEndCapture(stream, &graph);
     cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);
     graphCreated=true;
   }
cudaGraphLaunch(instance, stream);
cudaStreamSynchronize(stream);
}
```

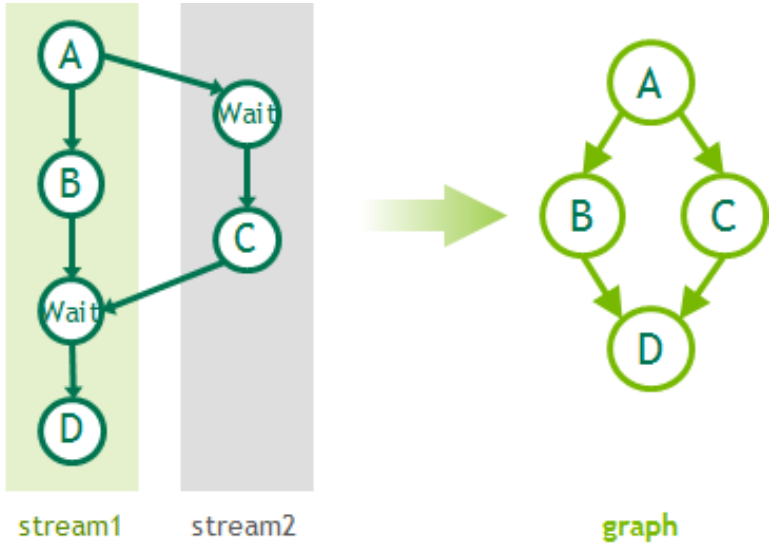
This is not actually executed, it just captures information about it

Effective time per kernel (including overheads), gives 3.4µs (vs 2.9µs kernel execution time), so we have successfully further reduced the overheads. Note that in this case, the time to create and instantiate the graph is relatively large around 400µs, (0.02µs per-kernel)

Creating a Graph Using Capture

CUDA Graphs can be used to describe operations involving multiple streams

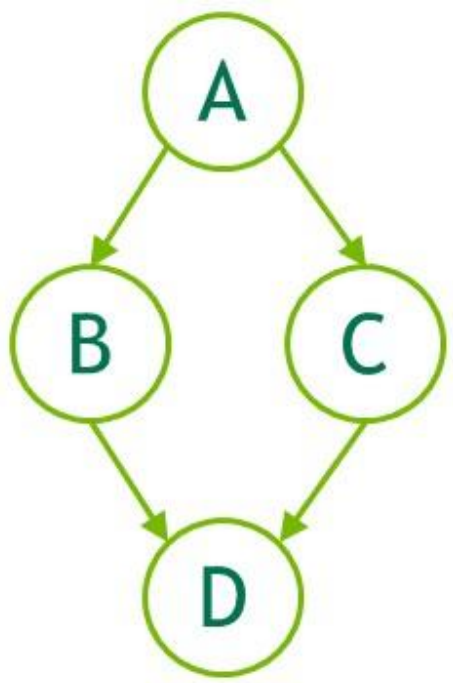
```
// Start by initating stream capture
cudaStreamBeginCapture(stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
}
```



When an event from a captured stream is waited on by another stream, this is set in capture mode if it is not already. When cross-stream dependencies are present in stream capture, *cudaStreamEndCapture()* must still be called in the same stream where *cudaStreamBeginCapture()* was called; this is the origin stream.

Creating a Graph Using Graph APIs

```
cudaGraph_t graph;
cudaGraphNode_t a,b,c,d;

// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// For the purpose of this example, we'll create
// the nodes separately from the dependencies to
// demonstrate that it can be done in two stages.
// Note that dependencies can also be specified
// at node creation.

cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams_a);
cudaGraphAddHostNode(&b, graph, NULL, 0,   &nodeParams_b);
cudaGraphAddMemcpyNode(&c, graph, NULL, 0, &nodeParams_c);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams_d);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);     // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);     // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);     // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);     // C->D
```



Three-Stage Execution Model

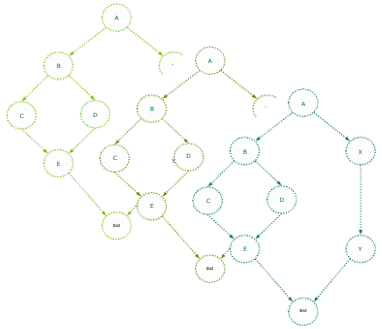
Minimizes Execution Overheads – Pre-Initialize As Much As Possible

Define

Single Graph "Template"

Created in host code or built up from libraries

Instantiate



Multiple "Executable Graphs"

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

Execute

Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

CUDA Graph Management API

Instantiate CUDA Graphs

```
__host__cudaError_t cudaGraphInstantiateWithFlags ( cudaGraphExec_t* pGraphExec, cudaGraph_t graph,
                                                    unsigned long long flags )

__host__cudaError_t cudaGraphInstantiate ( cudaGraphExec_t* pGraphExec, cudaGraph_t graph,
                                           cudaGraphNode_t* pErrorNode, char* pLogBuffer,
                                           size_t bufferSize )
```

- `pGraphExec` [OUT]: Returns instantiated graph
- `graph` [IN]: Graph to instantiate
- `flags` [IN]: Flags to control instantiation (`cudaGraphInstantiateFlagAutoFreeOnLaunch` | **`cudaGraphInstantiateFlagUseNodePriority`**).
- `pErrorNode` [OUT]: In case of an instantiation error, this may be modified to indicate a node contributing to the error
- `pLogBuffer` [OUT]: A character buffer to store diagnostic messages
- `bufferSize` [IN]: Size of the log buffer in bytes

Returns: `cudaSuccess`, `cudaErrorInvalidValue`

Three-Stage Execution Model

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

Define

Single Graph "Template"

Created in host code or built up from libraries

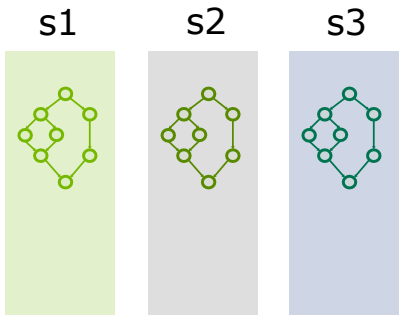
Instantiate

Multiple "Executable Graphs"

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

Execute



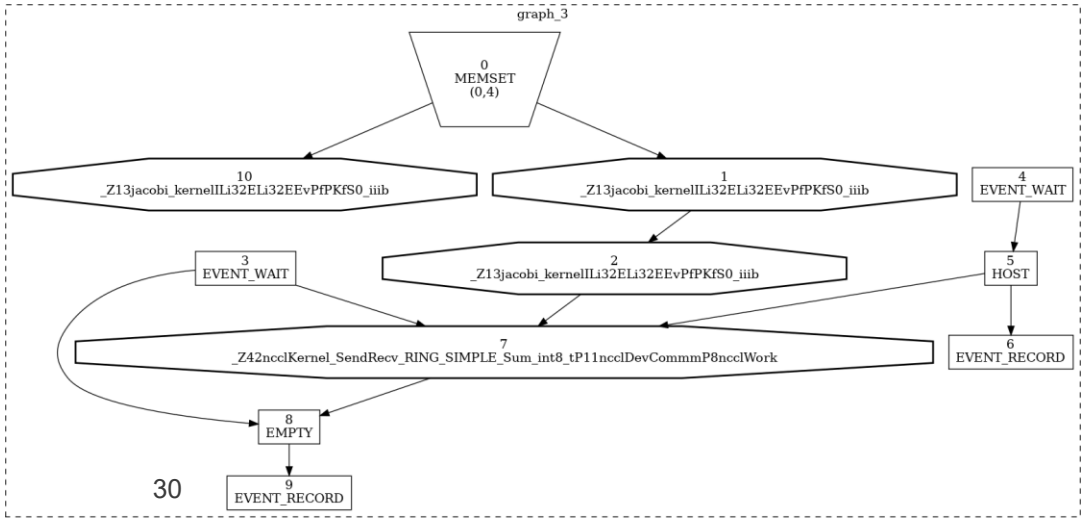
Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

New Execution Mechanism

Graphs Can Be Generated Once Then Launched Repeatedly

```
while (l2_norm > tol && iter < iter_max) {


    cudaGraphLaunch(graph_calc_norm_exec[iter%2],
                    compute_stream);
    cudaStreamSynchronize(compute_stream);
    MPI_Allreduce(l2_norm_h, &l2_norm, 1,
                  MPI_REAL_TYPE, MPI_SUM,
                  MPI_COMM_WORLD);
    l2_norm = std::sqrt(l2_norm);

    if (!csv && 0 == rank && (iter % 100) == 0) {
        printf("%5d, %0.6f\n", iter, l2_norm);
    }
}
```



Generated with

```
cudaGraphDebugDotPrint(graphs[calculate_norm][0],
                       "jacobi_graph.dot",0)
```

and

```
dot -Tpng jacobi_graph.dot -o jacobi_grap.png
```

CUDA Graph Management API

Free Resources

```
__host__cudaError_t cudaGraphDestroy ( cudaGraph_t graph )
```

- graph[IN]: Graph to destroy

Returns: cudaSuccess, cudaErrorInvalidValue

Destroys the graph specified by graph, as well as all of its nodes.

```
__host__cudaError_t cudaGraphExecDestroy ( cudaGraphExec_t graphExec )
```

- graphExec[IN]: Executable graph to destroy

Returns: cudaSuccess, cudaErrorInvalidValue

Destroys the executable graph specified by graphExec.

Solution

https://github.com/FZJ-JSC/tutorial-multi-gpu/tree/main/10-H_CUDA_Graphs_and_Device-initiated_Communication_with_NVSHMEM

Things to pay attention to

- On some iterations (e.g., even iterations) the code reads from ***a*** and writes in ***a_new***
- On other iterations (e.g., odd iterations) the code reads from ***a_new*** and writes in ***a***

Thus, we need two graphs, one for the even iterations, and one for the odd iterations

```
274          cudaGraphExec_t graph_exec[2];
```

Things to pay attention to

- On some iterations we do not calculate the norm
- On other iterations we do

Thus, we need two graphs, one for when we check the norm, and one for when we do not

```
275          cudaGraphExec_t graph_calc_norm_exec[2];
```

Graph Capture

```
283             cudaGraph_t graphs[2][2];
284             for (int iter = 0; iter < 4; ++iter)
285             {
286                 const bool calculate_norm = (iter < 2);
287                 //TODO: Begin capturing a graph in compute_stream
288                 CUDA_RT_CALL(cudaStreamBeginCapture(compute_stream, cudaStreamCaptureModeGlobal));
289                 CUDA_RT_CALL(cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream));
290                 CUDA_RT_CALL(cudaEventRecord(reset_l2norm_done, compute_stream));
291
292                 CUDA_RT_CALL(cudaStreamWaitEvent(push_stream, reset_l2norm_done, 0));
293                 launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm,
294                                      push_stream);
295
296                 launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, calculate_norm,
297                                      push_stream);
298                 CUDA_RT_CALL(cudaEventRecord(push_prep_done, push_stream));
299
300                 const int top = rank > 0 ? rank - 1 : (size - 1);
301                 const int bottom = (rank + 1) % size;
```

Graph Capture

```
303             // Apply periodic boundary conditions
304             NCCL_CALL(ncclGroupStart());
305             NCCL_CALL(ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream));
306             NCCL_CALL(ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream));
307             NCCL_CALL(ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream));
308             NCCL_CALL(ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream));
309             NCCL_CALL(ncclGroupEnd());
310             CUDA_RT_CALL(cudaEventRecord(push_done, push_stream));
311
312             launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1), nx, calculate_norm,
313                                  compute_stream);
314
315             if (calculate_norm) {
316                 CUDA_RT_CALL(cudaStreamWaitEvent(compute_stream, push_prep_done, 0));
317                 CUDA_RT_CALL(cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost,
318                                              compute_stream));
319             }
320
321             CUDA_RT_CALL(cudaStreamWaitEvent(compute_stream, push_done, 0));
322
323             const int is_even = iter % 2;
324             //TODO: End capturing `graphs[calculate_norm]+is_even` in compute_stream
325             CUDA_RT_CALL(cudaStreamEndCapture(compute_stream, graphs[calculate_norm]+is_even));
326
327             std::swap(a_new, a);
328         }
```

Graph Instances Creation

```
329         {
330             const bool calculate_norm = false;
331             //TODO: Instantiate graphs without norm calculation: What happens if cudaGraphInstantiateFlagUseNodePriority is **not** used?
332             //      - Instantiate `graphs[calculate_norm][0]` to `graph_exec+0`.
333             CUDA_RT_CALL(cudaGraphInstantiateWithFlags(graph_exec+0, graphs[calculate_norm][0], cudaGraphInstantiateFlagUseNodePriority));
334             //      - Instantiate `graphs[calculate_norm][1]` to `graph_exec+1`.
335             CUDA_RT_CALL(cudaGraphInstantiateWithFlags(graph_exec+1, graphs[calculate_norm][1], cudaGraphInstantiateFlagUseNodePriority));
336         }
337         {
338             const bool calculate_norm = true;
339             //TODO: Instantiate graphs with norm calculation:
340             //      - Instantiate `graphs[calculate_norm][0]` to `graph_calc_norm_exec+0`.
341             CUDA_RT_CALL(cudaGraphInstantiateWithFlags(graph_calc_norm_exec+0, graphs[calculate_norm][0], cudaGraphInstantiateFlagUseNodePriority));
342             //      - Instantiate `graphs[calculate_norm][1]` to `graph_calc_norm_exec+1`.
343             CUDA_RT_CALL(cudaGraphInstantiateWithFlags(graph_calc_norm_exec+1, graphs[calculate_norm][1], cudaGraphInstantiateFlagUseNodePriority));
344         }
345         // TODO: Destroy cudaGraph_t objects they are no longer required after instantiation
346         CUDA_RT_CALL(cudaGraphDestroy(graphs[0][0]));
347         CUDA_RT_CALL(cudaGraphDestroy(graphs[0][1]));
348         CUDA_RT_CALL(cudaGraphDestroy(graphs[1][0]));
349         CUDA_RT_CALL(cudaGraphDestroy(graphs[1][1]));
350     }
```

Graph Launch

```
376         while (l2_norm > tol && iter < iter_max) {
377             const bool calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);
378             if (calculate_norm) {
379                 // TODO: Launch `graph_calc_norm_exec[iter%2]` in `compute_stream`
380                 CUDA_RT_CALL(cudaGraphLaunch(graph_calc_norm_exec[iter%2], compute_stream));
381                 CUDA_RT_CALL(cudaStreamSynchronize(compute_stream));
382                 MPI_CALL(MPI_Allreduce(l2_norm_h, &l2_norm, 1, MPI_REAL_TYPE, MPI_SUM, MPI_COMM_WORLD));
383                 l2_norm = std::sqrt(l2_norm);
384 
385                 if (!csv && 0 == rank && (iter % 100) == 0) {
386                     printf("%5d, %0.6f\n", iter, l2_norm);
387                 }
388             } else {
389                 // TODO: Launch `graph_exec[iter%2]` in `compute_stream`
390                 CUDA_RT_CALL(cudaGraphLaunch(graph_exec[iter%2], compute_stream));
391             }
392             iter++;
393         }
```

Destruction

```
436         CUDA_RT_CALL(cudaGraphExecDestroy(graph_exec[1]));
437         CUDA_RT_CALL(cudaGraphExecDestroy(graph_exec[0]));
438         CUDA_RT_CALL(cudaGraphExecDestroy(graph_calc_norm_exec[1]));
439         CUDA_RT_CALL(cudaGraphExecDestroy(graph_calc_norm_exec[0]));
440
441         CUDA_RT_CALL(cudaEventDestroy(reset_l2norm_done));
442         CUDA_RT_CALL(cudaEventDestroy(push_done));
443         CUDA_RT_CALL(cudaEventDestroy(push_prep_done));
444         CUDA_RT_CALL(cudaStreamDestroy(push_stream));
445         CUDA_RT_CALL(cudaStreamDestroy(compute_stream));
446
447         CUDA_RT_CALL(cudaFreeHost(l2_norm_h));
448         CUDA_RT_CALL(cudaFree(l2_norm_d));
449
450         CUDA_RT_CALL(cudaFree(a_new));
451         CUDA_RT_CALL(cudaFree(a));
452
453         CUDA_RT_CALL(cudaFreeHost(a_h));
454         CUDA_RT_CALL(cudaFreeHost(a_ref_h));
455
456         NCCL_CALL(ncclCommDestroy(nccl_comm));
```

What else can we improve?

- We need to launch a new graph at each iteration, because comm happens on the CPU and then we have this extra sync at every iteration
- This is an issue especially for short running kernels
  - We have this loop kernel launch -> kernel execution -> wait for kernel, just because we have to communicate top/bottom rows by calling ncclSend/MPI_Send etc from the host
- It would be great if we could issue communication calls directly from within the kernels, so we can just enqueue kernels back-to-back and avoid that extra launch/sync overhead

CPU-Initiated Communication

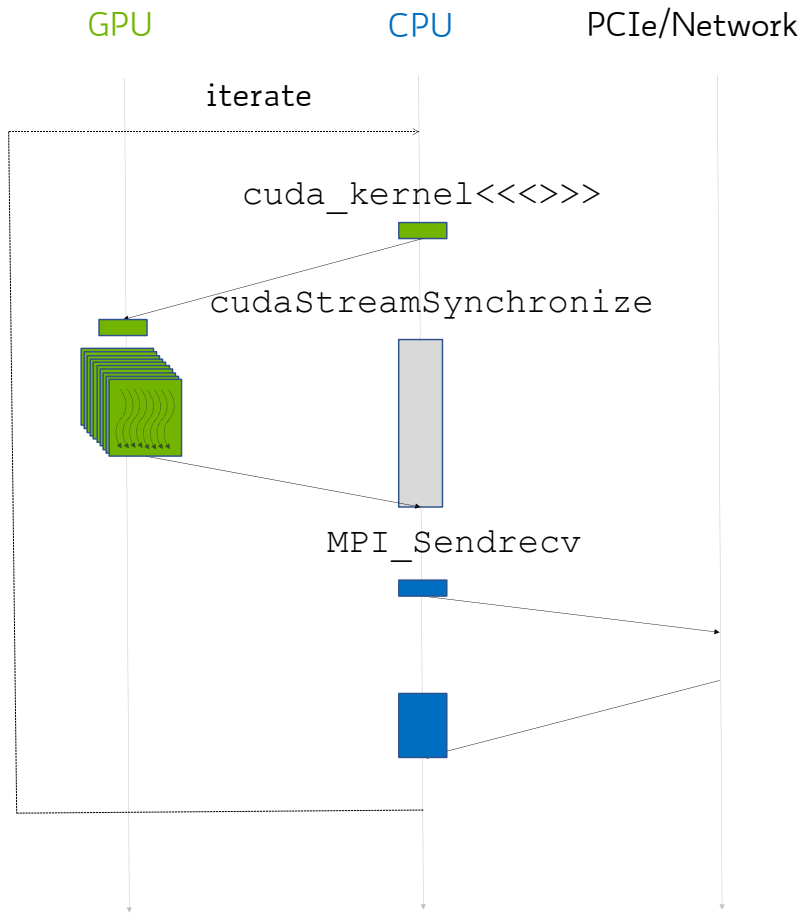
- Compute on GPU
- Communication from CPU

Synchronization at boundaries

Commonly used model, but –

- Offload latencies in critical path
- Communication is not overlapped

Hiding increased code complexity, not hiding limits strong scaling

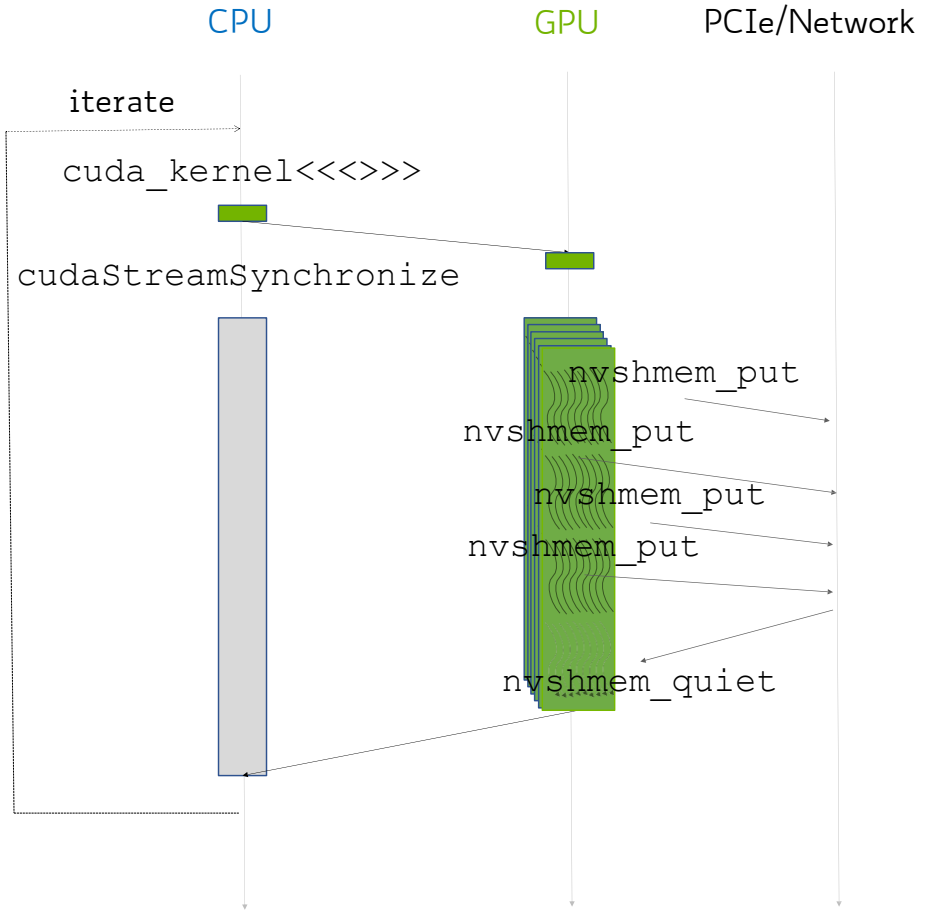


GPU-Initiated Communication

- Compute on GPU
- Communication from GPU

Benefits

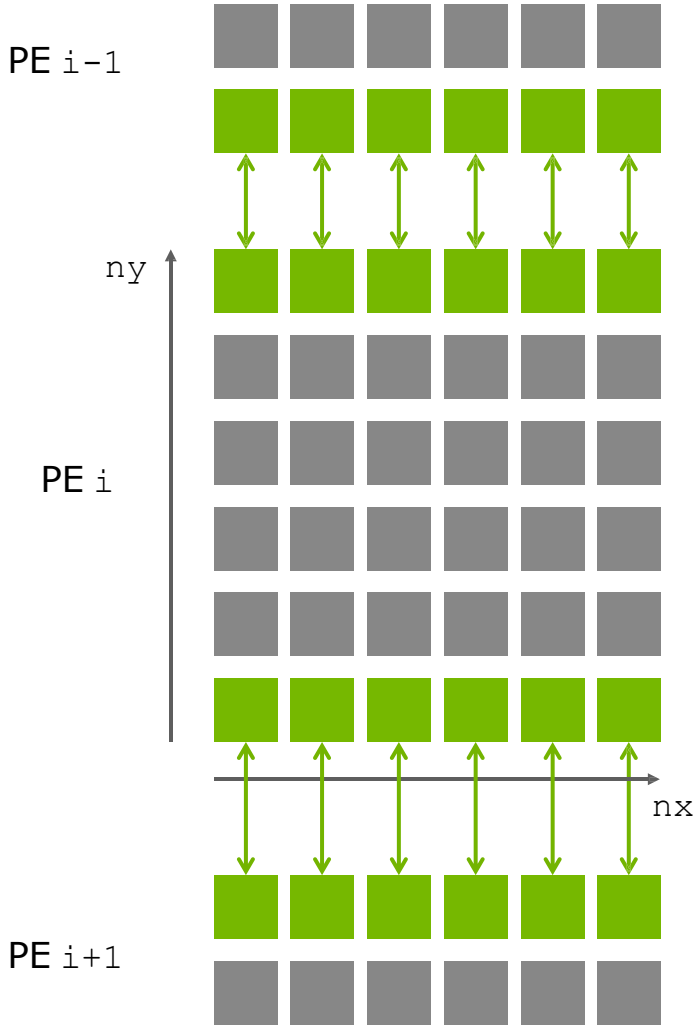
- Eliminates offloads latencies
- Compute and communication overlap by threading
- Easier to express algorithms with inline communication



Thread-Level Communication

- Allows fine grained communication and computation overlap
- Efficient mapping to NVLink fabric on DGX systems

```
__global__ void stencil_single_step(float *u, float *v, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  compute(u, v, ix, iy);
  // Thread-level data communication API

  if (iy == 1)
    nvshmem_float_p(u+(ny+1)*nx+ix, u[nx+ix], top_pe);
  if (iy == ny)
    nvshmem_float_p(u+ix, u[ny*nx+ix], bottom_pe);
}

int main(){
    ...
    for (int iter = 0; iter < N; iter++) {
        stencil_single_step<<<..., stream>>>(a, a_new, …);
        nvshmem_barrier_all_on_stream(stream);
        std::swap(a, a_new);
    }
    ...
}
```



Solution

https://github.com/FZJ-JSC/tutorial-multi-gpu/tree/main/10-H_CUDA_Graphs_and_Device-initiated_Communication_with_NVSHMEM

Host Code

```
a = (float *) nvshmem_malloc(nx*(chunk_size+2)*sizeof(float));
a_new = (float *) nvshmem_malloc(nx*(chunk_size+2)*sizeof(float));

…
while ( l2_norm > tol && iter < iter_max ) {

    …
    jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(
        a_new, a, l2_norm_d, iy_start, iy_end, nx,
        top, iy_end_top, bottom, iy_start_bottom );
    nvshmemx_barrier_all_on_stream(compute_stream);
     …
}
nvshmem_barrier_all();
nvshmem_free(a);
nvshmem_free(a_new);
```

Kernel

```
__global__ void jacobi_kernel( ... ) {
    const int ix = bIdx.x*bDim.x+tIdx.x;
    const int iy = bIdx.y*bDim.y+tIdx.y + iy_start;
    float local_l2_norm = 0.0;
    if ( iy < iy_end && ix >= 1 && ix < (nx-1) ) {
        const float new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                                     + a[ (iy+1) * nx + ix ] + a[ (iy-1) * nx + ix ] );
        a_new[ iy * nx + ix ] = new_val;
        if ( iy_start == iy )
            nvshmem_float_p(a_new +    top_iy*nx + ix, new_val, top_pe);
        if ( iy_end == iy )
            nvshmem_float_p(a_new + bottom_iy*nx + ix, new_val, bottom_pe);
        float residue = new_val - a[ iy * nx + ix ];
    }
    atomicAdd( l2_norm, local_l2_norm );
}
```

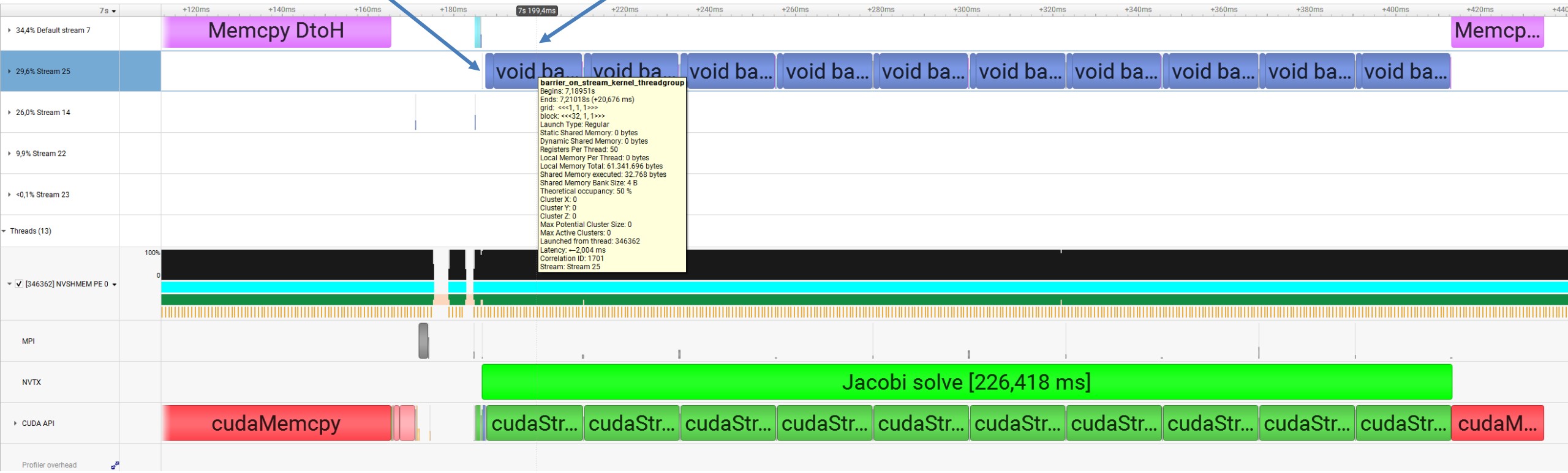
Performance

```
Num GPUs: 4.
16384x16384: 1 GPU:   6.2975 s, 4 GPUs:  22.5230 s, speedup:     0.28, efficiency:     6.99
```

Profile

Computation

Wait for communication

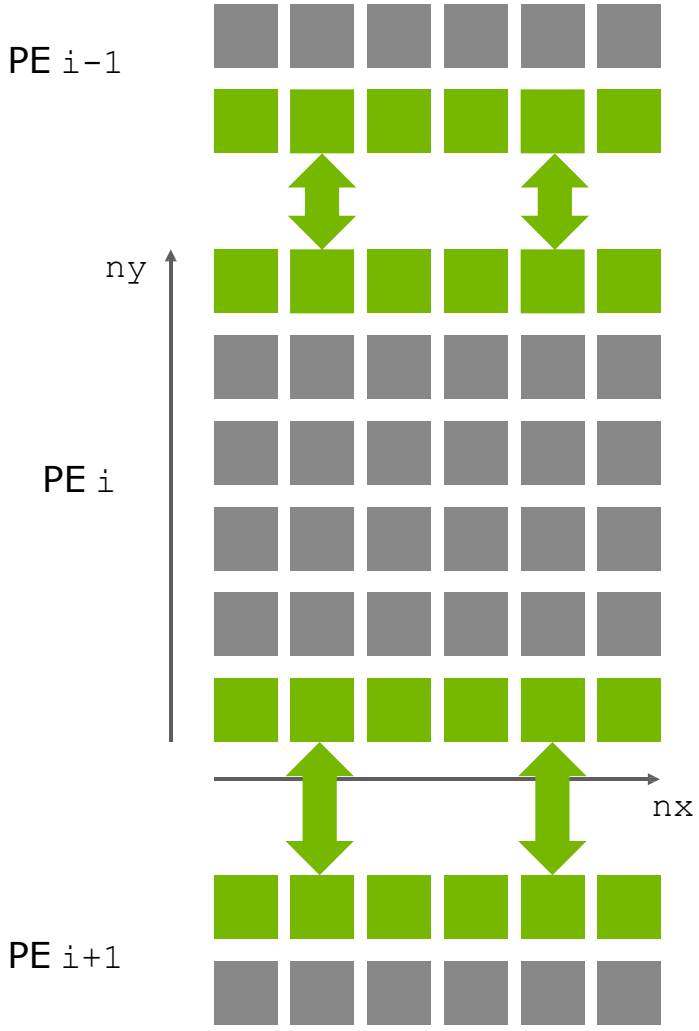


How to improve efficiency?

Thread-Group Communication

- NVSHMEM operations can be issued by all threads in a block/warp
- More efficient data transfers over networks like IB
- Still allows inter-warp/inter-block overlap

```
__global__ void stencil_single_step(float *u, float *v, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  compute(u, v, ix, iy);
  // Thread block-level communication API
  int bos = get_block_offet(blockIdx,blockDim);
  if (blockIdx.y == 0)
    nvshmemx_float_put_nbi_block(u+(ny+1)*nx+bos, u+nx+ bos, blockDim.x,top_pe);
  if (blockIdx.y == (blockDim.y-1))
    nvshmemx_float_put_nbi_block(u+bos, u+ny*nx+bos, blockDim.x, bottom_pe);
}
int main(){
    ...
    for (int iter = 0; iter < N; iter++) {
        stencil_single_step<<<..., stream>>>(a, a_new, …);
        nvshmem_barrier_all_on_stream(stream);
        std::swap(a, a_new);
    }
    ...
}
```

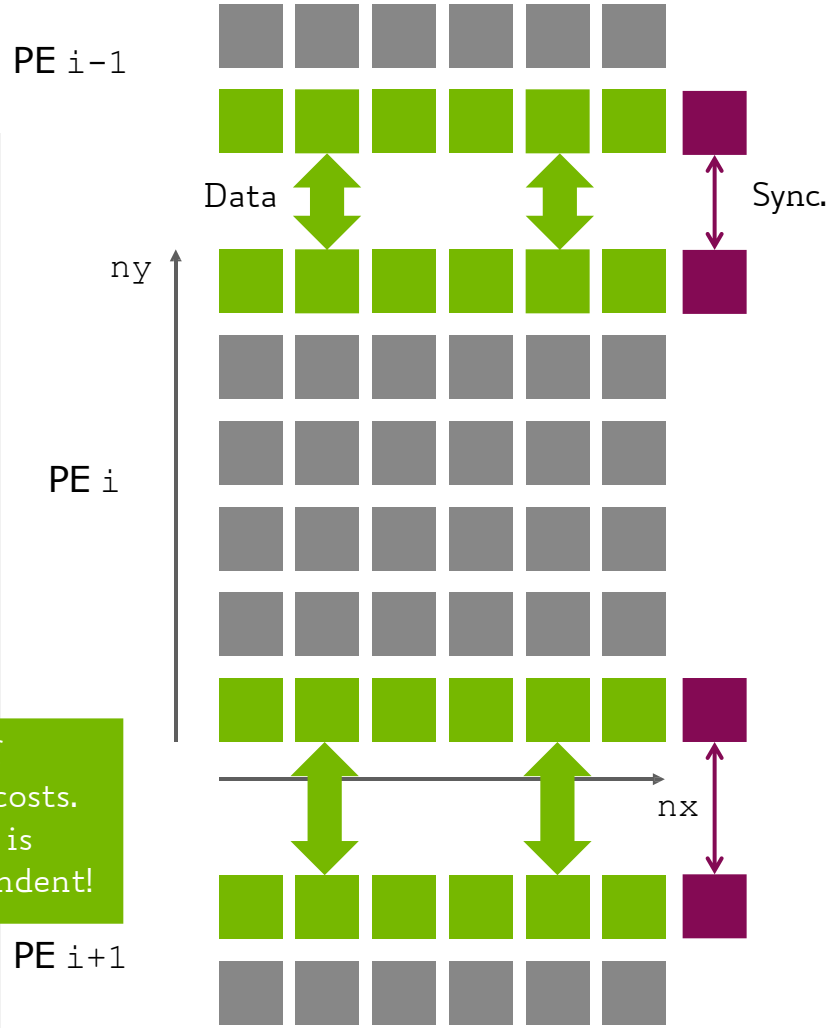


In-Kernel Synchronization

- Point-to-point synchronization across PEs within a kernel
- **Enables kernel fusion**

```
__global__ void stencil_multi_step(float *u, float *v, int N, int *sync, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  for (int iter = 0; iter < N; iter++) {
    swap(u, v);
    compute(u, v, ix, iy);
    // Thread block-level data exchange (assume even/odd iter buffering)
    int bos = get_block_offet(blockIdx,blockDim);
    if (blockIdx.y == 0)
      nvshmemx_float_put_nbi_block(u+(ny+1)*nx+bos, u+nx+bos, blockDim.x, top_pe);
    if (blockIdx.y == (blockDim.y-1))
      nvshmemx_float_put_nbi_block(u + bos, u+ny*nx+bos,  blockDim.x, bottom_pe);
    if (blockIdx.y == 0 || blockIdx.y == (blockDim.y-1)) {
      __syncthreads();
      nvshmem_quiet();
      if (threadIdx.x == 0 && threadIdx.y == 0) {
        nvshmem_atomic_inc(sync, top_pe);
        nvshmem_atomic_inc(sync, bottom_pe);
      }
    }
    nvshmem_wait_until(sync, NVSHMEM_CMP_GT, 2*iter*gridDim.x);
  }
}
```

Be aware of synchronization costs. Best strategy is application dependent!



Collective Kernel Launch

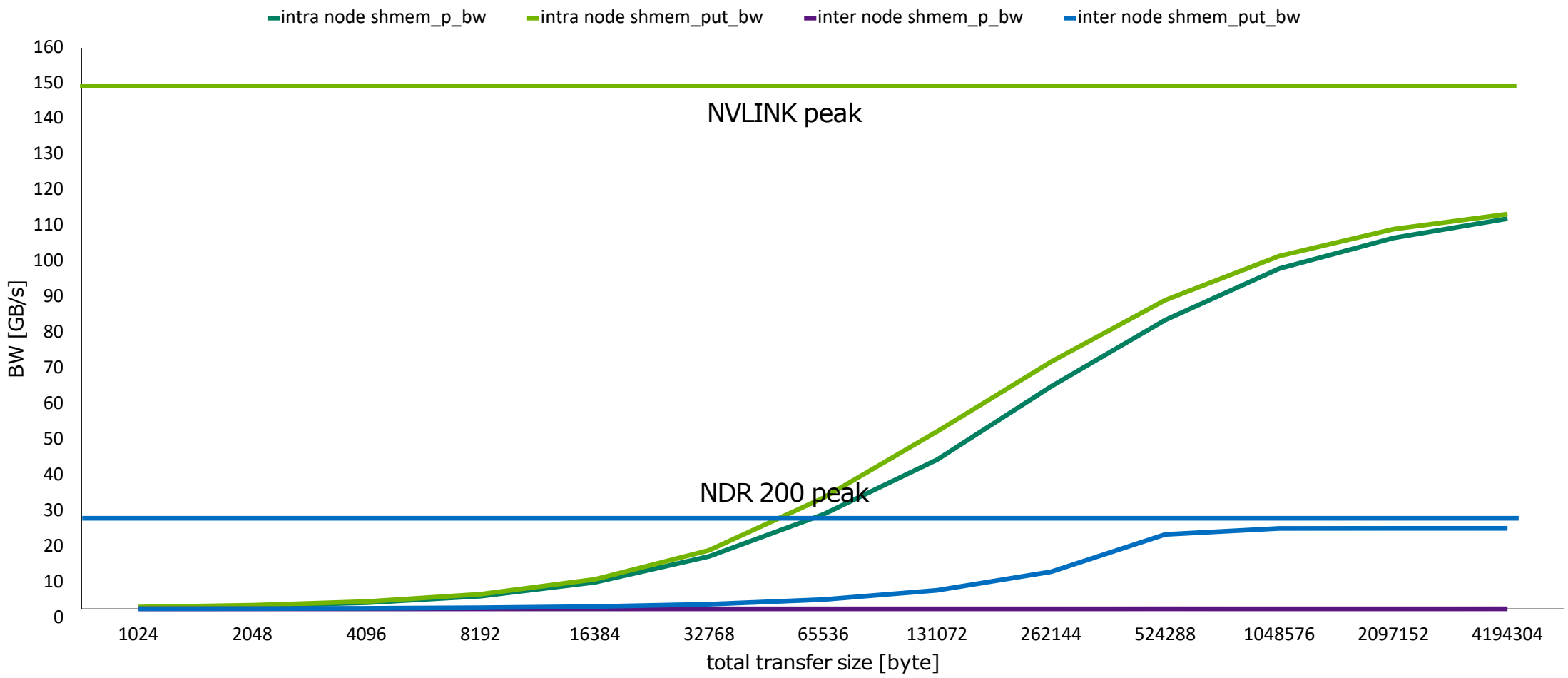
Ensures progress when using device-side inter-kernel synchronization

NVSHMEM Usage	CUDA Kernel launch
Device-Initiated Communication	Execution config syntax `<<<...>>>` or launch APIs
Device-Initiated Synchronization	`nvshmemx_collective_launch`

- CUDA's throughput computing model allows (encourages) grids much larger than a GPU can fit
- Inter-kernel (i.e., inter-GPU) synchronization requires producer and consumer threads to execute concurrently
- Collective launch guarantees co-residency using CUDA cooperative launch and requirement of 1PE/GPU

NVSHMEM Perftests

shmem_p_bw and shmem_put_bw on JEDI – NVIDIA GH200 120GB



Extra Infos/Perspectives

- NCCL also provides a device-side API (recently added)
- More details on how things work under the hood:
  - https://arxiv.org/abs/2409.09874

Dynamic Parallelism (and where it is used)

Slide courtesy: Tiziano De Matteis (Vrije University)

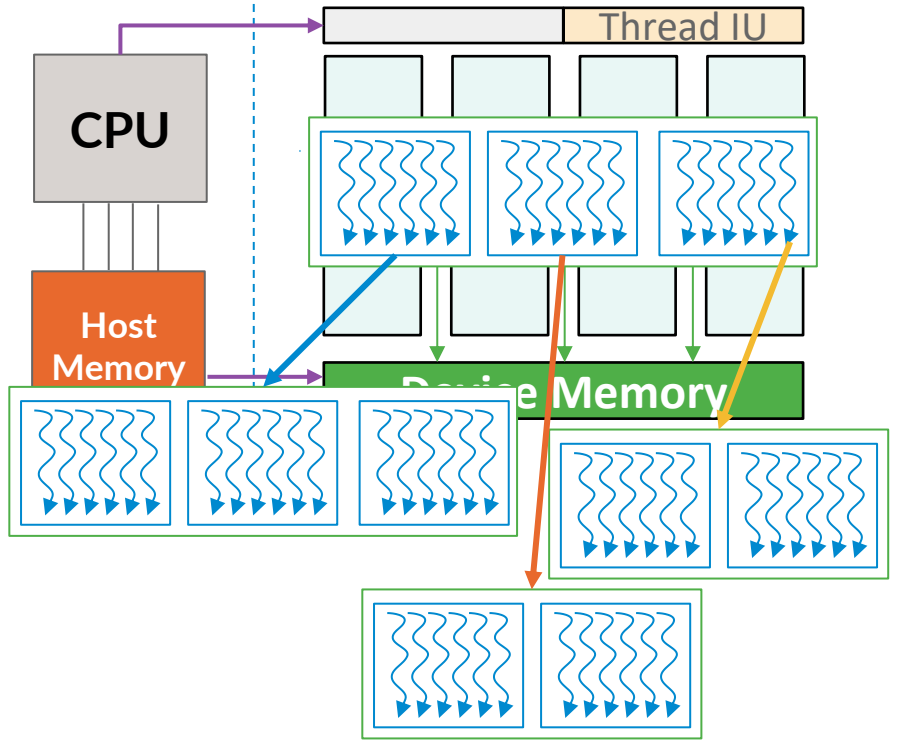
CUDA Dynamic Parallelism

So far we have seen how the host (and only the host) can instantiate grids and launch kernels

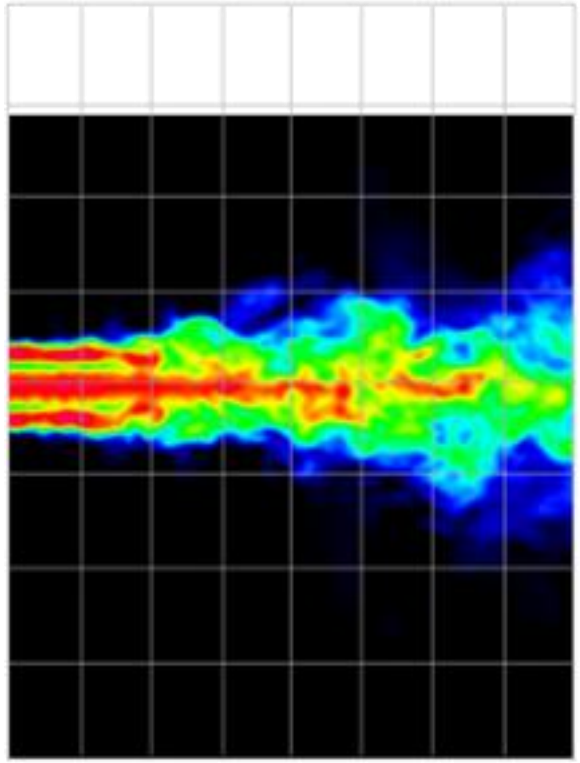
Starting from Kepler architecture (2012, CUDA Capability 3), NVIDIA GPUs support device-side kernel launches

CUDA Dynamic Parallelism refers to the ability of threads executing on the GPU to launch new grids

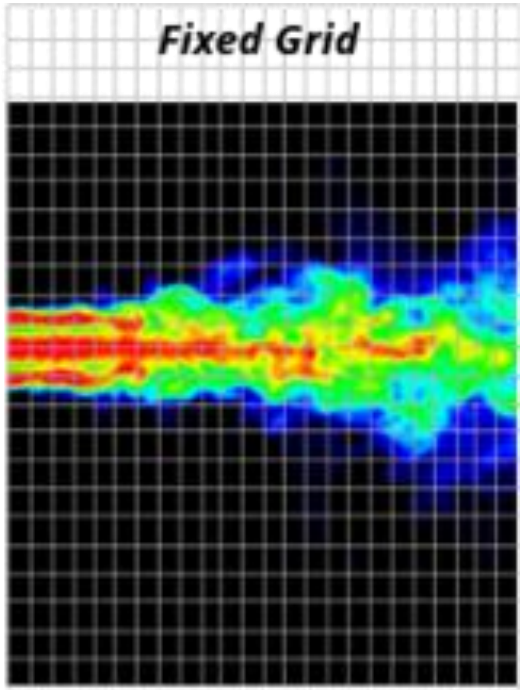
This enables **Nested Parallelism**, where threads discover more work that can be parallelized. This is extremely useful when the amount of work is unknown or imbalanced



EXAMPLE: TURBULENCE SIMULATION



Initial Grid



Static Fixed Grid

Dynamic Grid

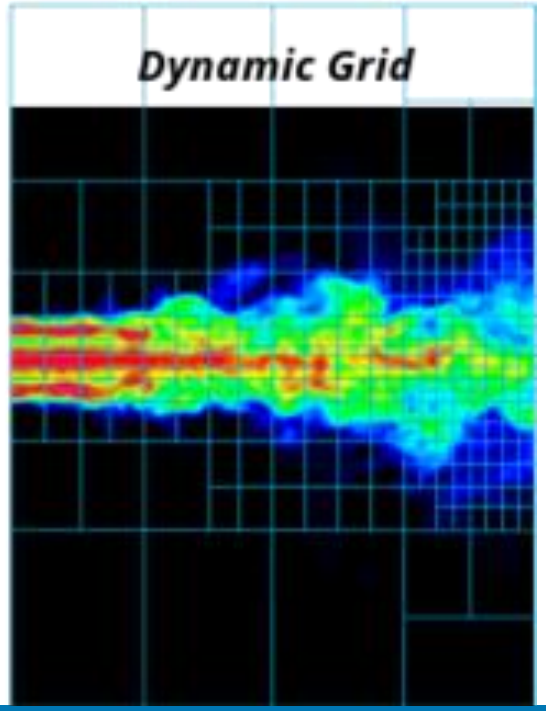
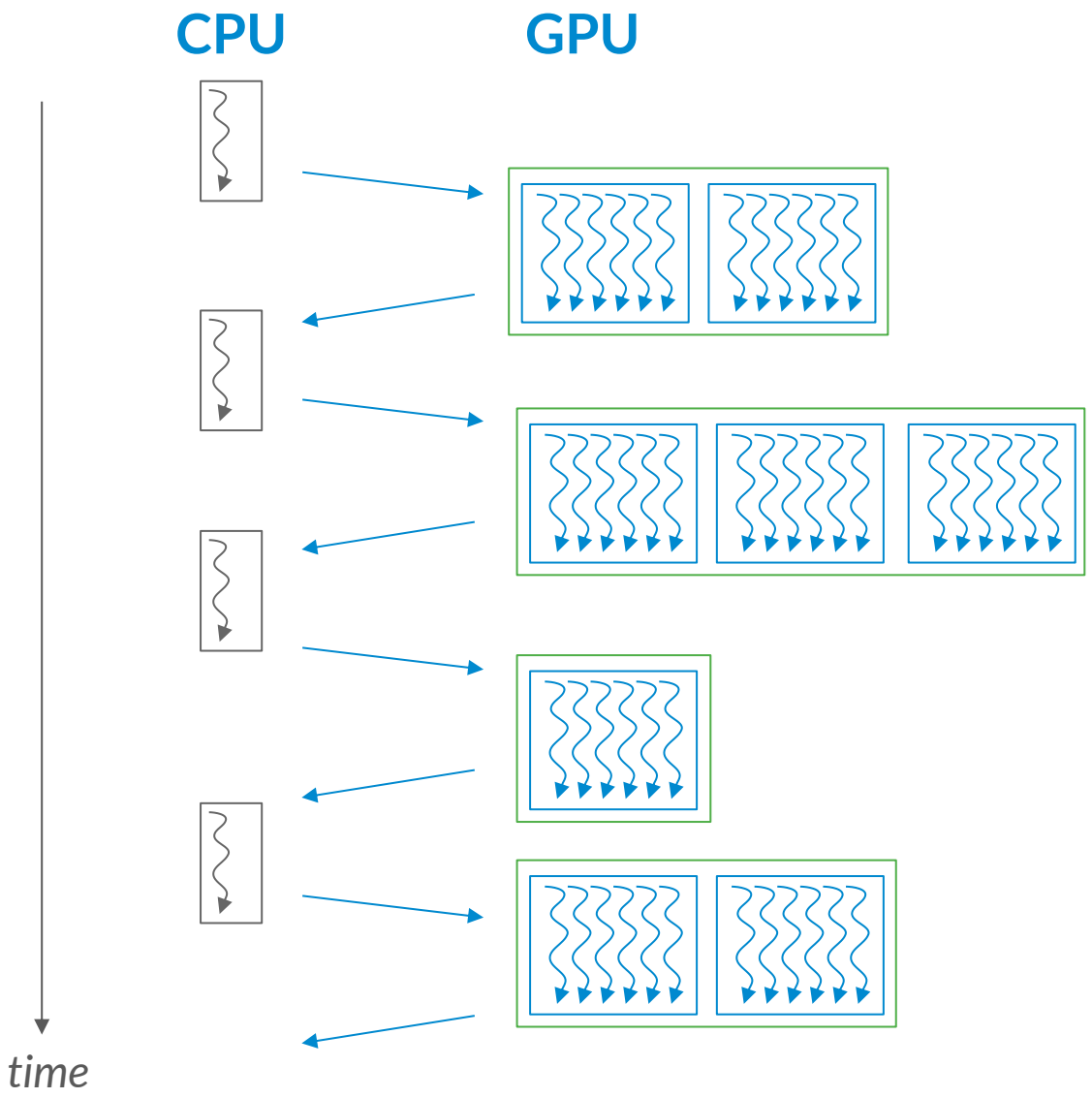
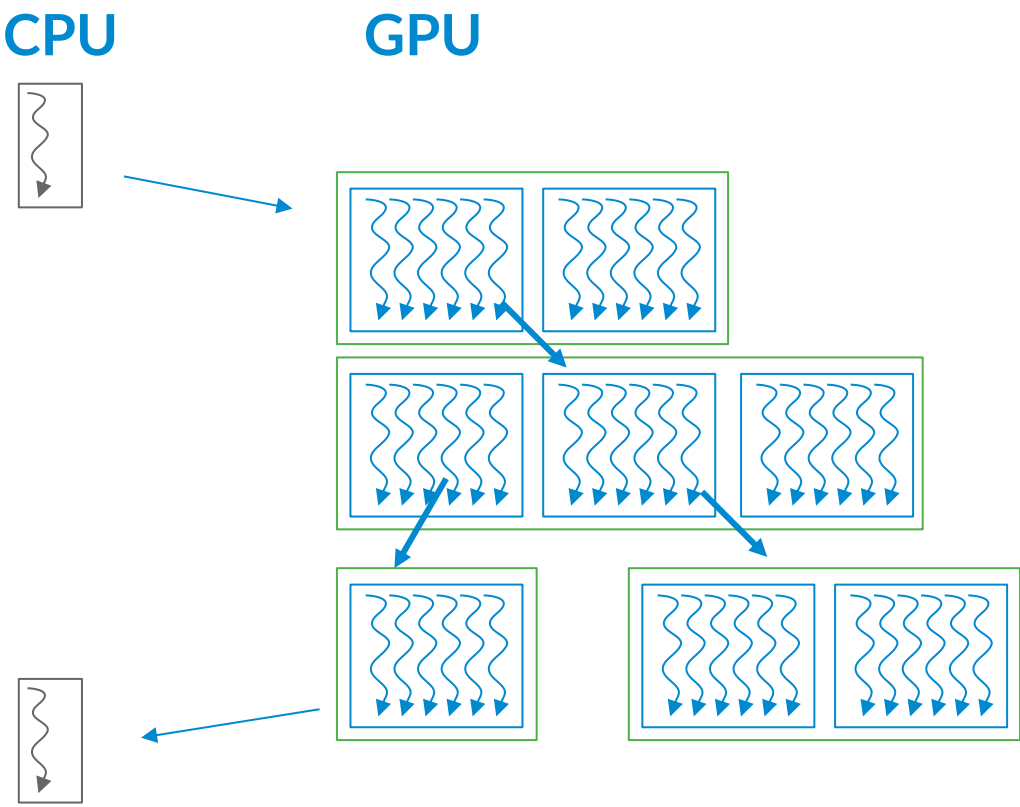


Figure from D. Kirk, W. W. Hwu, I. El Hajj ."Programming Massively Parallel Processors: A Hands-on Approach"

KERNEL LAUNCH WITH AND W/O DP

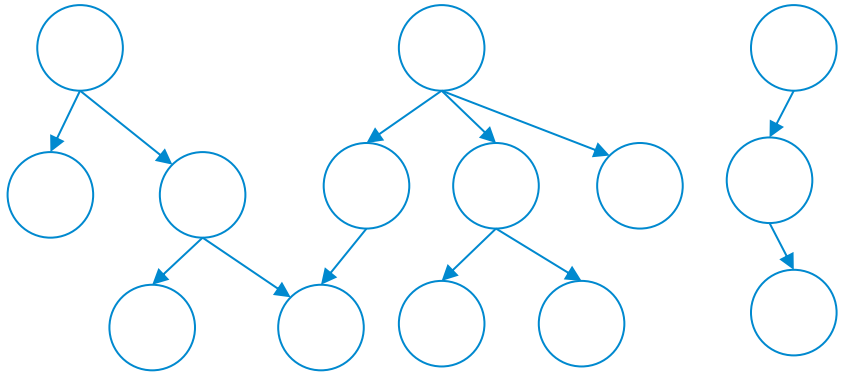


W/O Dynamic Parallelism



With Dynamic Parallelism

Applications of Dynamic Parallelism



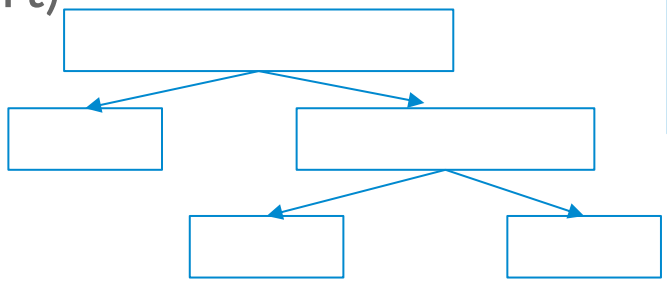
The amount of nested work is unknown.

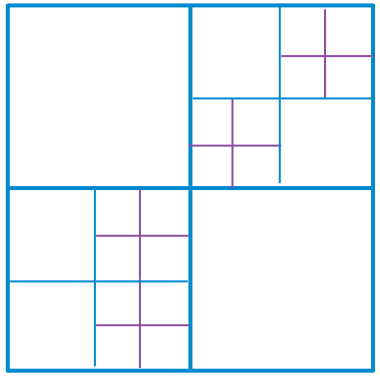
For instance it is **irregular**, varies across threads:

- E.g., graph algorithms (nodes have different # of neighbors)
- Bezier lines (each curve needs a different #points to draw)

Nested parallel work is **recursive** with unknown depth

- E.g., tree traversal (quadtree, octotree…)
- Divide and conquer algorithms (e.g., quicksort)





DYNAMIC PARALLELISM API

The device code to call a kernel is the same as the host code:

```
my_kernel<<<grid_dim, block_dim, 0, stream>>>(...)
```

In the “parent” kernel you can use `cudaDeviceSynchronize` to wait for the “child” grid to complete

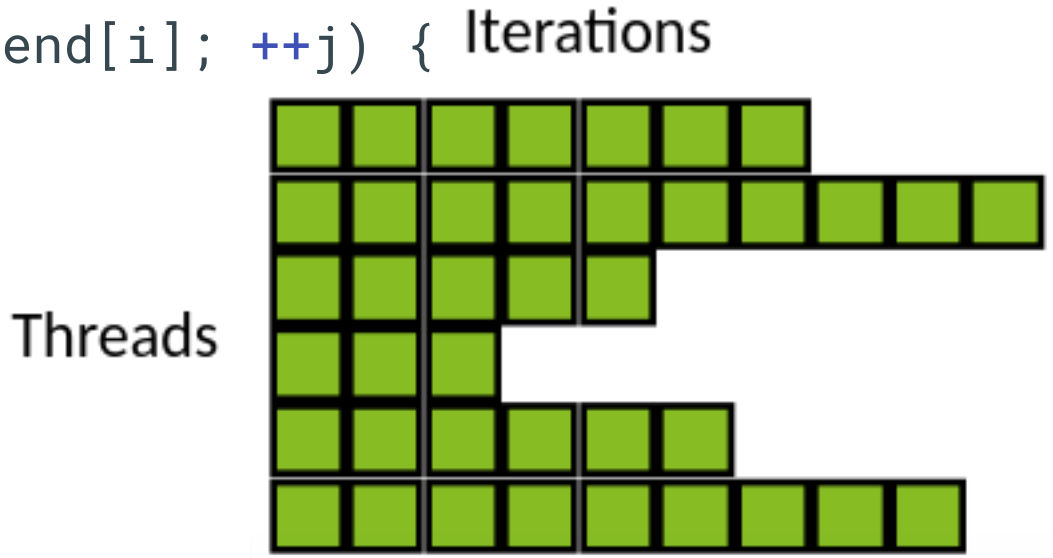
Memory is needed to buffer metadata for the pending launches (grid size, args, etc)

- By default it allows metadata for 2048 launches to be stored (the 2049th launch will fail)
- It can be changed must be done carefully (increases memory usage and overhead)
- If the launch is inside the kernel code and not conditioned, each thread that reaches that line will launch a child grid.

EXAMPLE

Q: What's the problem(s) with this code?

```
__global__ void kernel(unsigned int* start, unsigned int* end,
                                            float* someData, float* moreData)
{

  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  doSomeWork_findMoreWork(someData[i]);

  for(unsigned int j = start[i]; j < end[i]; ++j) {
    doMoreWork(moreData[j], i);
  }
}
```



EXAMPLE

With dynamic parallelism

```
__global__ void kernel_parent(unsigned int* start, unsigned int* end,
                              float* someData, float* moreData) {
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  doSomeWork(someData[i]);

  kernel_child<<<ceil((end[i]-start[i])/256.0), 256>>>(start[i], end[i], moreData);
}
```

The loop is now executed by a kernel rather than by a thread (i.e., each thread does just one operation)

Summary and Considerations on Dynamic Parallelism

Dynamic Parallelism ensures **better work balance,** and offers advantages in terms of **programmability**

Launching grids with a very small number of threads could lead to **severe underutilization of the GPU resources**

General recommendations:

- Launch grids with a large number of threads (and/or thread blocks)
- You may want to apply a **threshold**: only launch grid greater than X, and serialize the rest
- **Aggregate** launches: collect all the work (e.g., within a block) and have a single launch

PROGRAMMING OTHER GPUS/OTHER WAYS OF PROGRAMMING GPUS

AMD Software Ecosystem

AMD released in 2016 **ROCm**, its software stack for GPU programming

ROCm package provides **libraries and programming tools** for developing applications on AMD GPUs

HIP is the API you can use to program AMD's GPUs

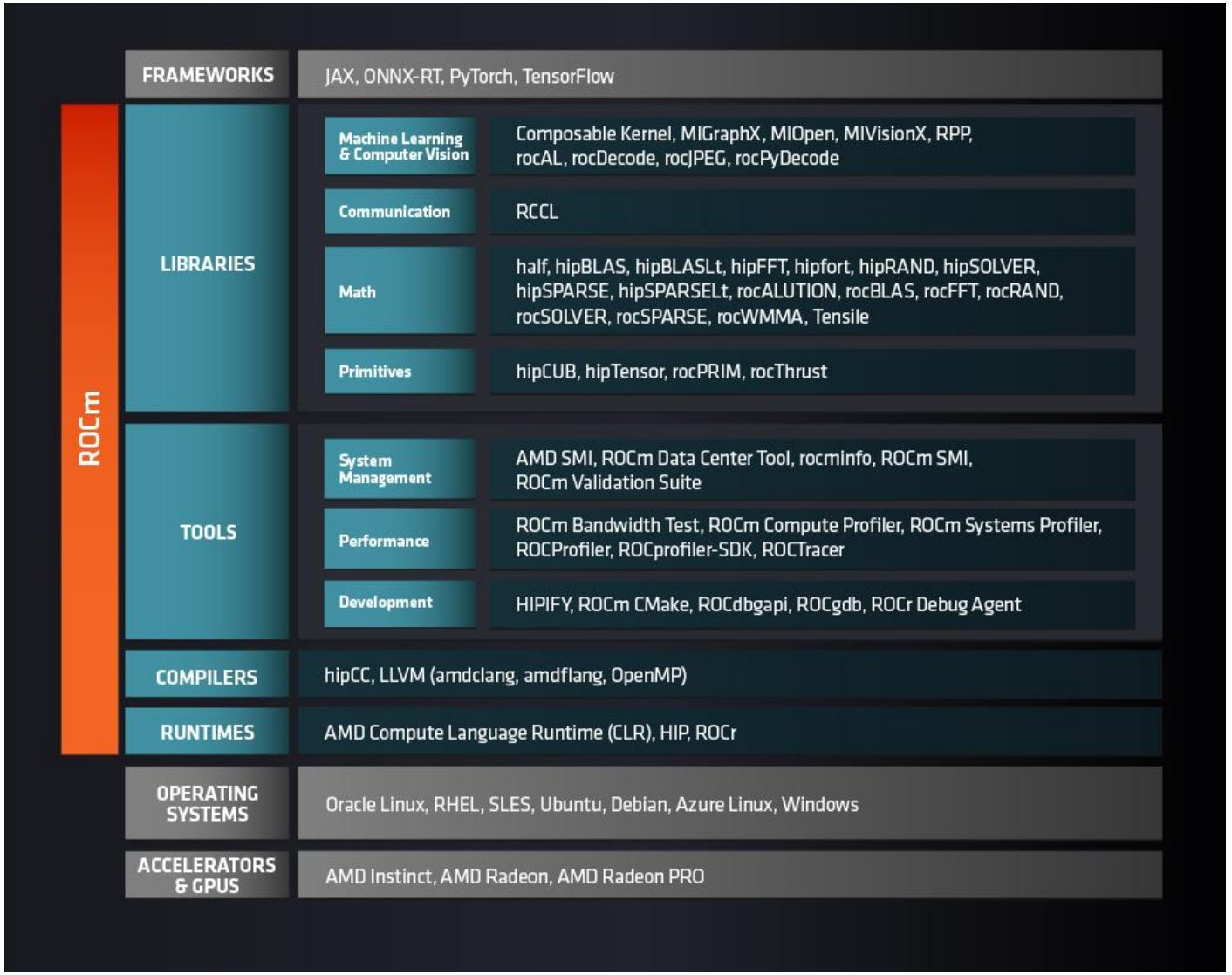
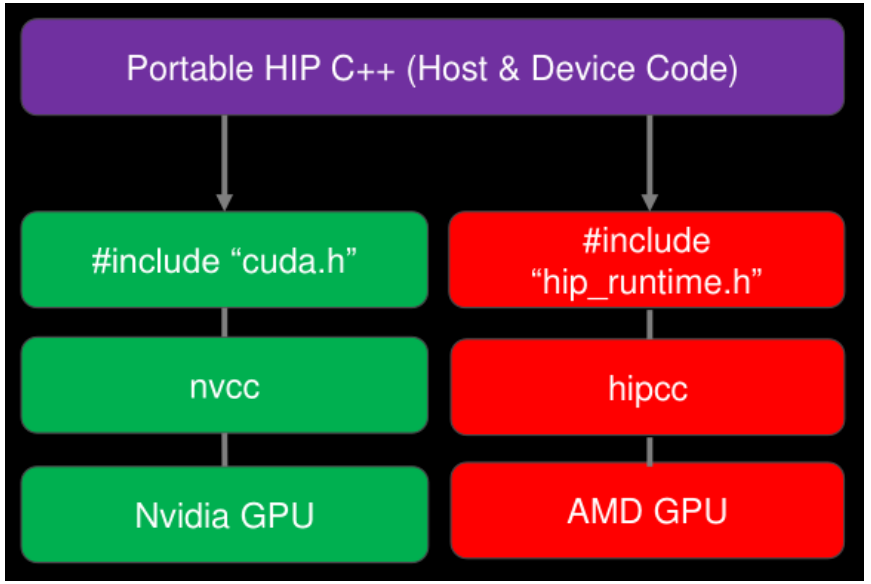


Image from AMD

HIP

Heterogeneous-Computing Interface for Portability (HIP) is a C++ dialect designed to let developers create portable applications that can run on AMD's accelerators as well as CUDA devices. It is **open-source**.



HIP can run on AMD GPUs (through hipCC compiler) and NVIDIA GPUs (through NVCC compiler)

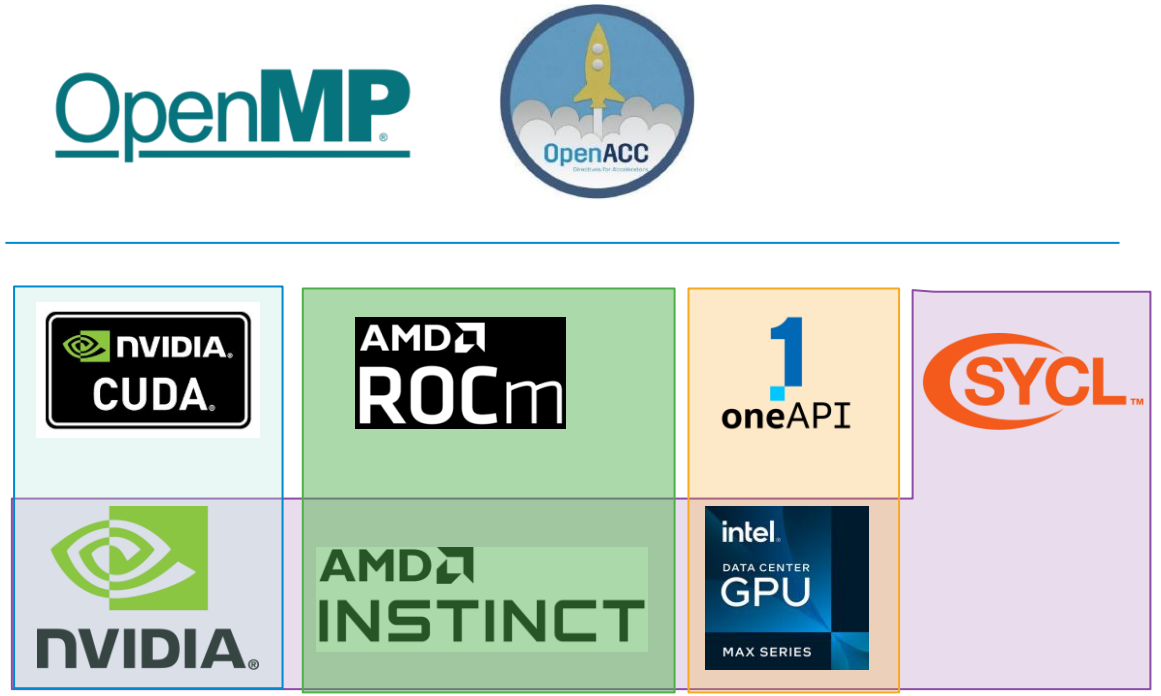
Syntactically similar to CUDA. Most CUDA API calls can be converted in place:

`cuda -> hip`

The HIPify tool automates much of the conversion work by performing a source-to-source transformation from CUDA to HIP

(High-Level) Portable GPU Programming

How to enable Code Portability across different GPU (vendors)?



High-Level Solutions

Low-Level API

Portable frameworks let you write once, run anywhere, avoiding being locked to a single vendor

OpenACC

OpenACC ("Open ACCelerators") is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. It provides a set of:

- Compiler directives (pragmas),
- Library routines
- Environment variables

to enable FORTRAN, C and C++ programs to execute on accelerator devices

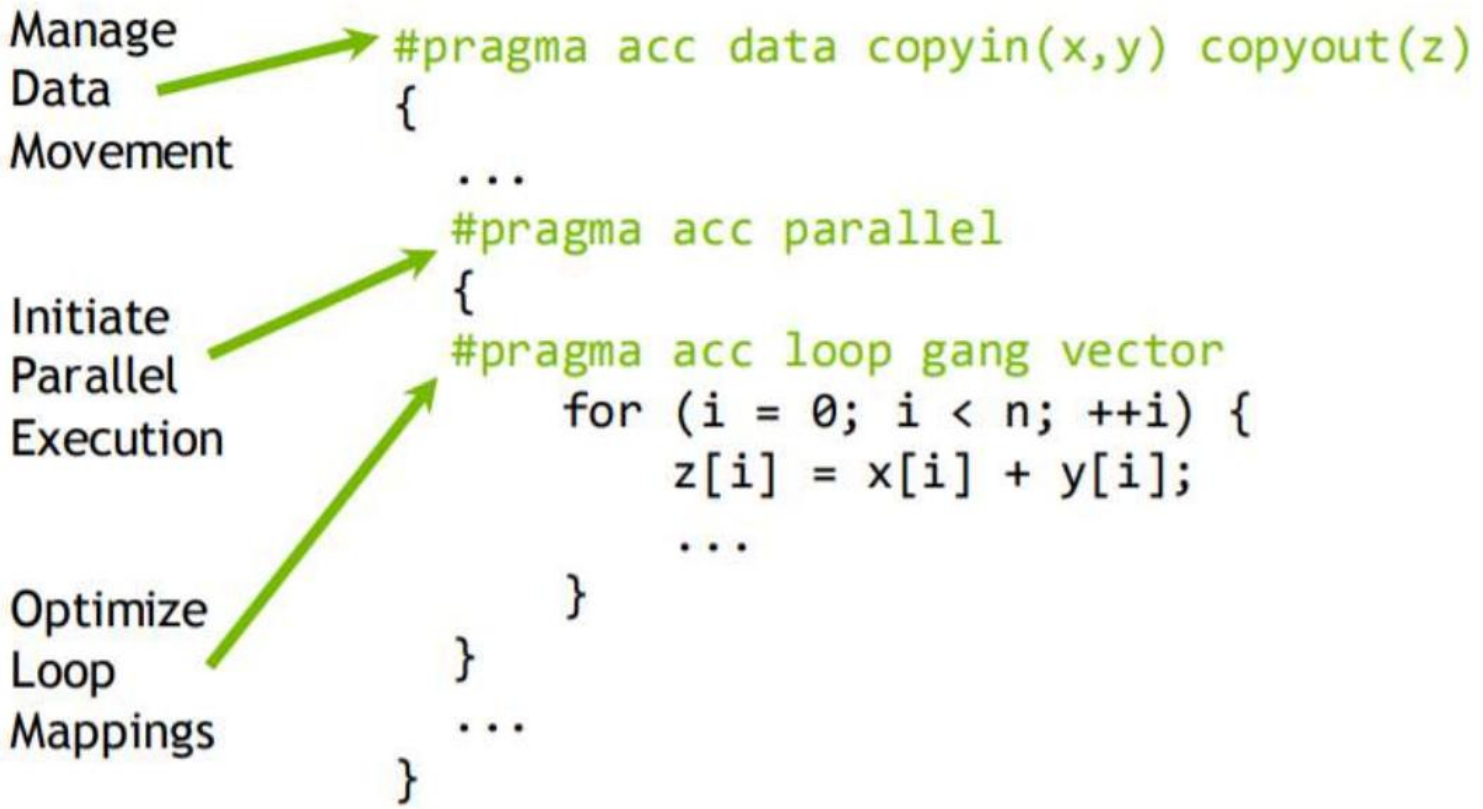
Similar to OpenMP, programmers can annotate source code to identify areas that should be accelerated.

In C/C++ the `#pragma` directive provides the compiler with information not specified in the language. For OpenACC they look like:

```
#pragma acc [the information goes here]
```

OPENACC

The OpenACC execution model still differentiate between code being executed on the host and on the device



WORKING WITH OPENACC

OpenACC programmers:

- Start with a sequential version
- Then annotate their program with directives
- Leaving most of the work to the compiler

The same code can run on different GPUs, and on CPU (also sequential)

Reality is different:

- Can be difficult to write code that works correctly and well w/ and w/o pragmas
- Some OpenACC programs behave differently if pragmas are ignored
- Strong performance dependency to the compiler (NVIDIA has better support to OpenACC)
- Limited control

Conclusions

Conclusions

- Due to the exponentially increasing computational demand, multi-GPU systems are necessary to achieve high performance and efficiency
- Achieving high performance requires optimization of data transfers, both between GPUs, but also between host and GPU
- To get a high performance you need to exploit overlapping, asynchronous calls, streams, graphs, and device-side operations
- This makes programming harder and code less readable
- High-level solutions exist, but they are still far from optimal

- Master thesis available on these and similar topics (I'll release an updated list on my personal page by (late) January)
- HPC is a large world, and we have seen just part of it
- Feedback on the course is appreciated