Programming Modern GPU Supercomputers

Jeff Hammond
Intel

Neither of these systems is has a many-core CPU or an NVIDIA GPU...

What programming model(s) take a developer from 2012 to 2022?
GPU Supercomputers are Multi-GPU Systems

Argonne Aurora: 2 CPU & 6 GPU

NERSC Perlmutter and ORNL Frontier: 1 CPU & 4 GPU

NVIDIA HGX A100: 2 CPU & 8 GPU


https://www.enterprisai.news/2020/05/20/amd-epyc-rome-tabbed-for-nvidias-new-dgx-but-hgx-has-intel-option/

1. Brief overview of SYCL ecosystem for GPUs.
2. SYCL w/ one device per process
   - MPI+X where X=SYCL
   - All of the good and bad of distributed-memory...
3. SYCL w/ multiple devices per process
   - More PGAS-like
   - All of the good and bad of shared-memory...

MPI = Message Passing Interface
PGAS = Partitioned Global Address Space
e.g. UPC and Fortran coarrays.
Overview of the SYCL™ Ecosystem for GPUs

  - oneAPI product compiler based on Clang/LLVM (open-source [1]).
  - Supports multiple HPC-oriented SYCL 2020 features including USM (pointers).
  - Supports Intel GPU, CPU, FPGA (by Intel) and NVIDIA (by CodePlay).

  - Product compiler (commercial support and free community edition).
  - Supports OpenCL™/SPIR-V devices (e.g. Intel GPU) and NVIDIA (via PTX).

- University of Heidelberg’s hipSYCL [https://github.com/illuhad/hipSYCL](https://github.com/illuhad/hipSYCL)
  - Based on Clang/LLVM, i.e. CUDA Clang (open-source).
  - Recently implemented SYCL 2020 USM [2,3].
  - Supports CPU (OpenMP), NVIDIA (CUDA) and AMD GPU (HIP/ROCm).

SYCL Ecosystem as of June 2020

Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL.
e.g. ROCm and CUDA
For more information: http://sycl.tech

Pure Distributed Memory Approach

- Portable: MPI works within and between nodes, VMs, etc.
- Homogeneous: Treat all inter-device relationships as if different nodes.
- Standard: MPI doesn’t know anything about GPUs (yet).
- Inefficient: Moves more data than necessary in multiple directions.
- Restrictive: \( \text{process\_per\_node} := \text{device\_per\_node} \times \text{proc\_per\_device} \).

1. SYCL moves device data to host
2. MPI moves data between hosts
3. SYCL moves host data to device
Improved Distributed Memory Approach

- Portable: MPI works within and between nodes, VMs, etc.
- Homogeneous: Treat all inter-device relationships as if different nodes.
- Standard: MPI doesn’t know anything about GPUs (yet).
- Inefficient: Moves more data than necessary in multiple directions.
- Restrictive: process_per_node := device_per_node * proc_per_device.
- Device support in MPI is non-standard and implementation-dependent.

1. Hosts make MPI calls with USM pointers (SYCL 2020)
### Intranode Shared-Memory Approach with MPI

- Portable: MPI works within and between nodes, VMs, etc.
- Homogeneous: Treat all inter-device relationships as if different nodes.
- Standard: MPI doesn’t know anything about GPUs (yet).
- Inefficient: Moves more data than necessary in multiple directions.
- Restrictive: `process_per_node := device_per_node * proc_per_device`.
- **MPI-3 shared-memory requires a special allocator** and other “fun”...

1. SYCL moves device data to host
2. SYCL moves host data to device

*C++ apps can use std::pmr or placement new...*
Intraneode Shared-Memory Approach with SYCL-Next?

- Portable: Use SYCL to copy data between different processes.
- Homogeneous: Treat all inter-device relationships as if different nodes.
- Standard: SYCL needs to know about multi-process contexts.
- Efficient: Moves only the data necessary.
- Restrictive: $\text{process}_{\text{per}_\text{node}} := \text{device}_{\text{per}_\text{node}} \times \text{proc}_{\text{per}_\text{device}}$.
- This feature does not currently exist... (it exists in proprietary models)

1. SYCL moves device data to device
Many HPC codes design for MPI first, and intranode parallelism second:

- There is more MPI parallelism available than intranode:
  - MPI ~100K cores, OpenMP ~100 cores, SIMD ~10 lanes
  - MPI ~5K GPUs, GPUs ~10, GPU coarse ~100, GPU fine ~50
- Intranode parallelism is growing; system size is not. Time to change priorities?

MPI remains an extremely good parallel model but moves too much data within the node (CPU and GPU).

- Upside of data privatization has historically outweighed downside of extra copies.
- MPI routinely beats OpenMP for multicore execution, when both exist, in part because most applications do a terrible job MPI+OpenMP. (#pragma omp parallel for is an anti-pattern!)
- GPUs change the relative costs of fork-join, intra/inter-device data movement, etc.
- Newer HPC workloads, e.g. deep learning training, are communication-intensive.

MPI isn’t synonymous with HPC anymore.
CUDA Example

nstream.h

```c
__global__ void nstream(
    int length,
    double s,
    double * A,
    double * B,
    double * C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    A[i] += B[i] + s * C[i];
}
```

nstream.cu

```c
const int bytes = length * sizeof(double);

check( cudaMalloc((void**)&d_A, bytes) );
check( cudaMalloc((void**)&d_B, bytes) );
check( cudaMalloc((void**)&d_C, bytes) );

dim3 DB(blockSize, 1, 1);
dim3 DG(length/blockSize, 1, 1);

nstream<<<DG, DB>>>(length, scalar, d_A, d_B, d_C);
check( cudaDeviceSynchronize() );
```
**OpenCL Example**

**nstream.cl**

```c
__kernel void nstream(
    int length,
    double s,
    __global double * A,
    __global double * B,
    __global double * C)
{
    int i = get_global_id(0);
    A[i] += B[i] + s * C[i];
}
```

**nstream.cpp**

```cpp
cl::Context gpu(CL_DEVICE_TYPE_GPU, &err);
cl::CommandQueue queue(gpu);

cl::Program program(gpu,
    prk::opencl::loadProgram("nstream.cl"), true);

auto kernel = cl::make_kernel<int, double, cl::Buffer,
    cl::Buffer, cl::Buffer>(program, "nstream", &err);

auto d_a = cl::Buffer(gpu, begin(h_a), end(h_a));
auto d_b = cl::Buffer(gpu, begin(h_b), end(h_b));
auto d_c = cl::Buffer(gpu, begin(h_c), end(h_c));

kernel(cl::EnqueueArgs(queue, cl::NDRange(length)),
    length, scalar, d_a, d_b, d_c);
queue.finish();
```
SYCL 1.2.1 Example

```cpp
sycl::gpu_selector device_selector;
sycl::queue q(device_selector);

sycl::buffer<double> d_A { h_A.data(), h_A.size() };  
sycl::buffer<double> d_B { h_B.data(), h_B.size() };  
sycl::buffer<double> d_C { h_C.data(), h_C.size() };

q.submit([&](sycl::handler& h) {
    auto A = d_A.get_access<sycl::access::mode::read_write>(h);
    auto B = d_B.get_access<sycl::access::mode::read>(h);
    auto C = d_C.get_access<sycl::access::mode::read>(h);

    h.parallel_for<class nstream>(sycl::range<1>{n}, [=] (sycl::id<1> it) {
        int i = it[0];
        A[i] += B[i] + s * C[i];
    });
});
q.wait();
```

Retains OpenCL's ability to easily target different devices in the same thread.

Accessors create DAG to trigger data movement and represent execution dependencies.

Parallel loops are explicit like C++ vs. implicit in OpenCL.

Kernel code does not need to live in a separate part of the program.
```cpp
sycl::queue q(gpu_selector{});

auto A = sycl::malloc_shared<double>(n, q);
auto B = sycl::malloc_shared<double>(n, q);
auto C = sycl::malloc_shared<double>(n, q);

q.submit([&](sycl::handler& h) {
    h.parallel_for(sycl::range<1>{n}, [=] (sycl::id<1> it) {
        int i = it[0];
        A[i] += B[i] + s * C[i];
    });
});
q.wait();

sycl::free(A, q);
sycl::free(B, q);
sycl::free(C, q);

Pointers: everyone's favorite footgun!

Lambda names are optional, but potentially useful for debugging.
```
SYCL 2020 Example

```cpp
sycl::queue q(gpu_selector{});

auto A = sycl::malloc_shared<double>(n, q);
auto B = sycl::malloc_shared<double>(n, q);
auto C = sycl::malloc_shared<double>(n, q);

q.parallel_for(sycl::range<1>{n}, [=] (sycl::id<1> it) {
    int i = it[0];
    A[i] += B[i] + s * C[i];
});
q.wait();

sycl::free(A, q);
sycl::free(B, q);
sycl::free(C, q);
```

Eliminate unnecessary syntax for expressing kernels.
auto H = sycl::malloc_host<double>(n, q);
// initialize H
auto D = sycl::malloc_device<double>(n, q);
q.memcpy(D, H, n*sizeof(double)).wait();
// do something with D on the device
q.memcpy(H, D, n*sizeof(double)).wait();
sycl::free(D, q);
// do something with H on the host
sycl::free(H, q);
**Multi-GPU SYCL Programming Concepts**

Similar to MPI...
- SYCL has explicit state: devices, queues, contexts...
- SYCL is asynchronous and provides fine and coarse grain synchronization.
- *Must solve domain decomposition and other application-specific problems.*

Unlike MPI...
- “...all the member functions and special member functions of the SYCL classes are thread safe.” [SYCL 1.2.1]
- Can choose between shared-memory (USM shared), PGAS (USM memcpy) and STL (buffer::copy) data management schemes.
private:
    std::vector<sycl::queue> list;
public:
    queues(void) {
        auto platforms = sycl::platform::get_platforms();
        for (auto & p : platforms) {
            auto devices = p.get_devices();
            for (auto & d : devices) {
                if (d.is_gpu()) {
                    list.push_back(sycl::queue(d));
                }
            }
        }
    }

Assumptions and logic to ensure devices are in the same context are hidden...
Buffers are wonderfully opaque but makes reasoning about sharing hard.

- USM shared data moves between host and device (d2d is not portable).
- USM device data does not migrate – start here.

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Initial Location</th>
<th>Accessible By</th>
<th>Migratable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>device</td>
<td>host</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Another device</td>
<td>Optional (P2P)</td>
</tr>
<tr>
<td>host</td>
<td>host</td>
<td>host</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any device</td>
<td>Yes (~PCIe)</td>
</tr>
<tr>
<td>shared</td>
<td>host / device / Unspecified</td>
<td>host</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Another device</td>
<td>Optional (P2P)</td>
</tr>
</tbody>
</table>

Data Models
Summary #2

- Like MPI, SYCL is explicit about providing a handle to state.
  - MPI communicators, groups, requests, windows, etc.
  - SYCL platforms, devices, queues, contexts, etc.
- Explicit device handles make the task of multi-GPU programming easier.
  - Contrast: CUDA runtime API hides the device id and CUBLAS contexts don't capture it...
- If GPU compute is tightly coupled, build a data-centric abstraction to manage allocation, data movement, synchronization, and collective compute.
  - PETSc, Elemental and Global Arrays are good examples of this in the MPI plane.
- Load-store is not a good abstraction for most interconnects...

```c
for (int i=0; i<ngpus; ++i) {
    check( cudaSetDevice(i) );
    check( cublasCreate(&contexts[i]) );
}

for (int i=0; i<ngpus; ++i) {
    check( cudaSetDevice(i) );
    check( cudaDeviceSynchronize() );
}
```
Where should we go from here?*

- Independent of SYCL, we need to redesign applications for multi-GPU nodes.
  - A side-effect of this is that we should get very good single-node performance without a dependency on any form of multi-processing.

- Future versions of SYCL should support more device-to-device features.
  - Industry standards must capture the characteristics of multiple vendors' hardware.
  - All HPC-oriented GPU vendors are doing something to support this, but it will take a year or two to understand what is universal.

- Device-to-device should not be limited to a single node.
  - We want to support MPI within SYCL device code. Prototyping in progress.

* Jeff’s opinion, which may not be shared by Intel or Khronos.