DS 7347
High-Performance Computing (HPC) and Data Science
Session 29

Robert Kalescky
Adjunct Professor of Data Science
HPC Research Scientist

August 2, 2022

Research and Data Sciences Services
Office of Information Technology
Center for Research Computing
Southern Methodist University

Outline

Assignments and Project

Session Questions

Distributed Computation

NVSHMEM

Assignments and Project

Project

- Completed project committed to project repo by August 4, 2022.
- On August 4, 2022 present for 10 minutes your work improving the performance of your workflow.

Session Questions

Session Questions

Thursday

What's special about RDMA?

Tuesday

What are some benefits and detriments to using GPUs?

Thursday

Generally, what types of computations benefit from using GPUs?

Heterogeneous Computing

- Heterogeneous computing is the use of hardware that is specialized to a particular task
- Enables hardware based acceleration of programs for specific areas of their code
- Devices are commonly referred to accelerators
- Examples:
    - Cryptographic accelerators
    - Field-programable gate arrays (FPGAs)
    - Digital signal processors (DSPs)
    - Graphics processing units (GPUs)

Heterogeneous Computing

- General-purpose GPU computing or GPGPU computing is the use of a GPU (graphics processing unit) to do general purpose scientific and engineering computing
- Model for GPU computing is to use a CPU and GPU together for heterogeneous co-processing computing
    - Sequential part of the application runs on the CPU
    - Computationally-intensive part is accelerated by the GPU
- From the user's perspective, the application just runs faster because it is using the high-performance of the GPU to boost performance

Terminology

host	The node.
device	The accelerator (GPUs).
host memory	The node's memory.
device memory	The accelerator's (GPU's) memory.
kernels	GPU function launched by the host and executed on the device
device function	GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

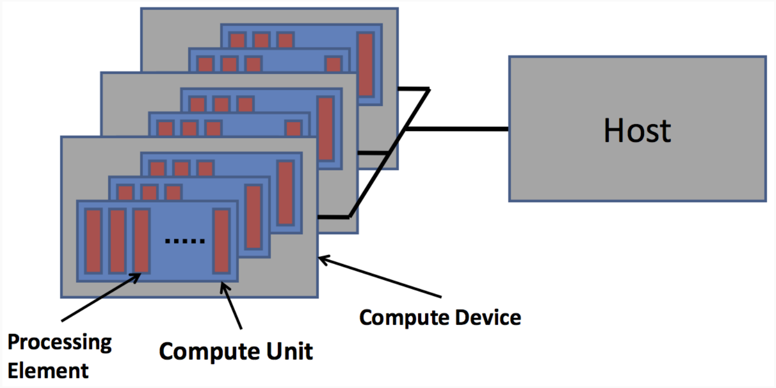
Heterogeneous Architectures

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight (very little creation overhead)
  - GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few

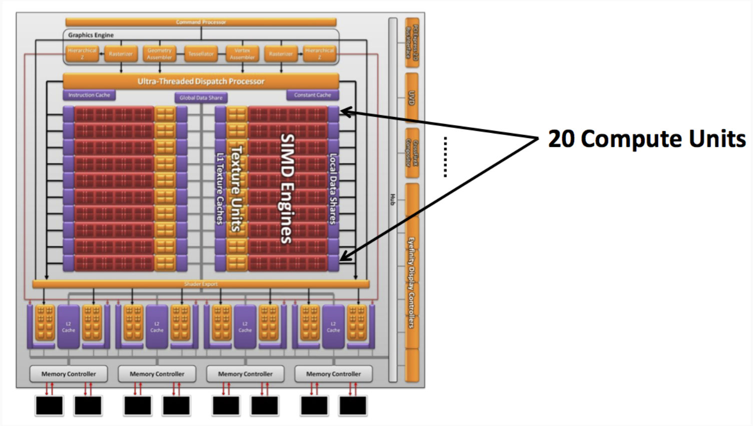
Accelerator Kernels

- A kernel is executed as a grid of thread blocks
    - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other
    - Synchronizing their execution
    - Hazard-free shared memory accesses
    - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

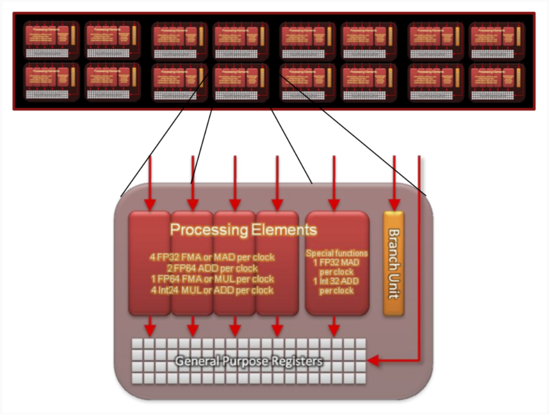
Host and Devices



Compute Units



Prcessing Elements


Processing Elements
4 FP32 FMA or MAD per clock
2 FP64 ADD per clock
1 FP64 FMA or MUL per clock
4 Int24 MUL or ADD per clock
Special functions
1 FP32 MAD
per clock
1 Int 32 ADD
per clock
Branch Unit
General Purpose Registers

Heterogeneous APIs

Libraries

- There are many libraries available that implement commonly used functions on GPGPUs.

OpenMP & OpenACC

- These are pragma and directive based tools use to decorate code to provide instructions to compiles as to what and how sections of code can be run on an accelerator.
- OpenACC and OpenMP each began their work independently, but have since begun the work of integrating their standards as they are very similar.

Heterogeneous APIs

CUDA

- Popular C-based programming language to program exclusively NVIDIA GPUs
- It has many features that make programming GPUs easier than OpenCL

OpenCL

- OpenCL (Open Compute Language) is a C-based programming language
- Hardware agnostic
- Supports a wide variety of heterogeneous platforms
- Heterogeneous Programming Tools

Heterogeneous APIs

There are a handful of high-level languages that allow for GPU-based calculations. Access to the GPU can be through function libraries or directly through lower level languages.

Kernel Interfaces

- PyOpenCL, PyCUDA, MATLAB, and Mathematica
- Kernels themselves are still written in OpenCL or CUDA specific languages

Numba

- Numba is a Python just-in-time compiler that can uses decorators to instruct the interpreter when to compile blocks of code.
- The compilation can be done for CPUs and GPUs.

NVIDIA CUDA

- CUDA is NVIDIA’s low level language for programming NVIDIA GPUs
- Available in in three language versions, CUDA C, CUDA C++, and CUDA Fortran
- Theses languages are similar to standard C, C++, and Fortran, but with implicit array or vector programming
- Low-level control over memory management and algorithms

OpenCL

- OpenCL is a vendor neutral standard low level language for programming many types of parallel devices
    - CPUs, GPUs, DSPs, and FPGAs
- Available in two language versions OpenCL C and OpenCL C++
    - OpenCL C also has C++ bindings
- Theses languages are similar to standard C, C++, and Fortran, but with implicit array or vector programming
- Low-level control over memory management and algorithms

CUDA vs. OpenCL

- CUDA is designed specifically for NVIDIA hardware
  - Has special knowledge and therefore CUDA code is more performance portable across NVIDIA hardware
- OpenCL is designed to be source portable across a wide variety of hardware
  - Not necessarily performance portable

NVIDIA CUDA C

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

OpenCL

```
__kernel void
vectorAdd(__global double *a, __global double *b, __global double *c, const
↪  unsigned int n)
{
    //Get our global thread ID
    int id = get_global_id(0);

    //Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

OpenACC

```c
#include <stdlib.h>

void saxpy(int n, float a, float *x, float *restrict y) {
   #pragma acc parallel loop
   for (int i = 0; i < n; ++i)
      y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv) {
   int N = 1<<20; // 1 million floats
   if (argc > 1) N = atoi(argv[1]);
   float *x = (float*)malloc(N * sizeof(float));
   float *y = (float*)malloc(N * sizeof(float));
   for (int i = 0; i < N; ++i) {
      x[i] = 2.0f;
      y[i] = 1.0f;
   }
   saxpy(N, 3.0f, x, y);
   return 0;
}
```

Numba

```python
@cuda.jit
def f(a, b, c):
    # like threadIdx.x + (blockIdx.x * blockDim.x)
    tid = cuda.grid(1)
    size = len(c)

    if tid < size:
        c[tid] = a[tid] + b[tid]
```

Numba

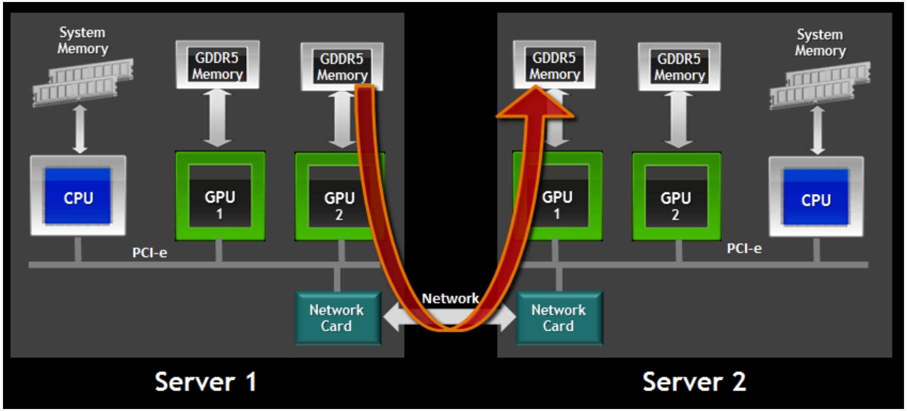
```python
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B."""
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Distributed Computation

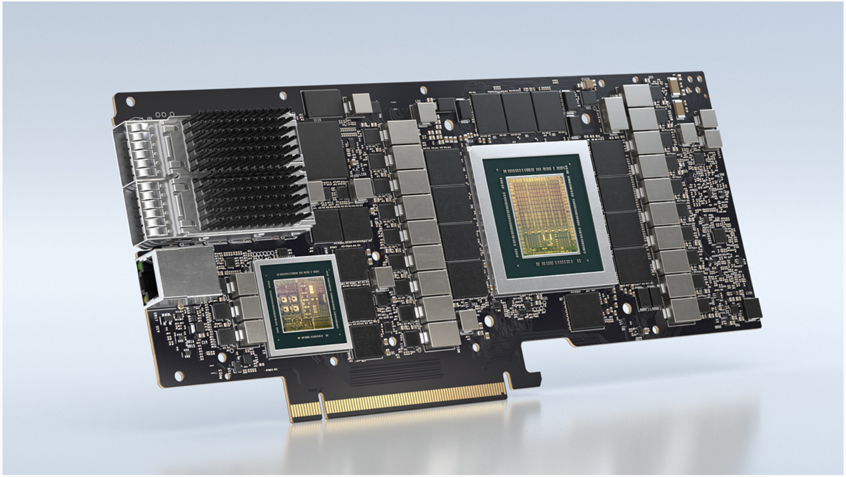
Very Low-Level Tools for Distributed Computations

- Unified Communication X (UCX)
- NVIDIA Mellanox MXM
- InfiniBand Verbs

Remote Direct Memory Access (RDMA)



Specialized Hardware for Specialized Use Cases



Low-Level Tools for Distributed Computations

- Message Passing Interface (MPI) (UCX)
- NVIDIA Collective Communication Library (NCCL) (UCX, NVLink)
- OpenSHMEM (UCX)
- NVSHMEM (UCX, NVLink)
- Apache Spark (UCX)

Low-Level Tools for Distributed Computations

- Message Passing Interface (MPI) (UCX)
- NVIDIA Collective Communication Library (NCCL) (UCX, NVLink)
- OpenSHMEM (UCX)
- NVSHMEM (UCX, NVLink)
- Apache Spark (UCX)

High-Level Tools for Distributed Computations

- mpi4py (MPI)
- Ray (MPI)
- Dask (MPI, Ray, Spark)
- Horovod (MPI, Ray, Spark)

Horovod

- Goal is to make it easy to take a single-GPU training script and successfully scale it to train across many GPUs in parallel.
- Optimized functions built using standard tools, such as MPI.
- Reimplements or wraps core functions of other frameworks.
    - TensorFlow
    - Keras
    - PyTorch
    - MXNet

Ray

- Goal is to make it easy to take a single-GPU training script and successfully scale it to train across many GPUs in parallel.
- Optimized functions built using standard tools, such as MPI.
- Reimplements or wraps core functions of other frameworks.
    - TensorFlow
    - PyTorch
    - Horovod (Automatically builds cluster)

NVSHMEM

Motivation for NVSHMEM

- Programming model for efficient and scalable multi-GPU codes.
- Based on OpenSHMEM.
- Provides a partitioned global address space (PGAS) that spans the memory of all included GPUs.
- Communication is initiated directly from the GPUs, which can be more performant than CPU-bound distributed programming models.

Estimating π

```
#include <iostream>
#include <curand_kernel.h>

#include <nvshmem.h>
#include <nvshmemx.h>

inline void CUDA_CHECK (cudaError_t err) {
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(err));
        exit(-1);
    }
}

#define N 1024*1024
```

Listing 1: Includes and definitions.

Estimating π

```
__global__ void calculate_pi(int* hits, int seed) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // Initialize random number state (unique for every thread in the grid)
    int offset = 0;
    curandState_t curand_state;
    curand_init(seed, idx, offset, &curand_state);

    // Generate random coordinates within (0.0, 1.0]
    float x = curand_uniform(&curand_state);
    float y = curand_uniform(&curand_state);

    // Increment hits counter if this point is inside the circle
    if (x * x + y * y <= 1.0f) {
        atomicAdd(hits, 1);
    }
}
```

Listing 2: Monte Carlo method for estimating π CUDA kernel.

Estimating π

```
int main(int argc, char** argv) {
    // Initialize NVSHMEM
    nvshmem_init();

    // Obtain our NVSHMEM processing element ID and number of PEs
    int my_pe = nvshmem_my_pe();
    int n_pes = nvshmem_n_pes();

    // Each PE (arbitrarily) chooses the GPU corresponding to its ID
    int device = my_pe;
    CUDA_CHECK(cudaSetDevice(device));
```

Listing 3: Initialize NVSHMEM and get local processing element (PE) ID and global number of PEs.

Estimating π

```
// Allocate host and device values
int* hits = (int*) malloc(sizeof(int));
int* d_hits = (int*) nvshmem_malloc(sizeof(int));

// Initialize number of hits and copy to device
*hits = 0;
CUDA_CHECK(cudaMemcpy(d_hits, hits, sizeof(int), cudaMemcpyHostToDevice));
```

Listing 4: Allocate memory on hosts and devices and initialize the number of hits.

Estimating π

```
    // Launch kernel to do the calculation
    int threads_per_block = 256;
    int blocks = (N / n_pes + threads_per_block - 1) / threads_per_block;

    int seed = my_pe;
    calculate_pi<<<blocks, threads_per_block>>>(d_hits, seed);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Accumulate the results across all PEs
    int* d_hits_total = (int*) nvshmem_malloc(sizeof(int));
    nvshmem_int_sum_reduce(NVSHMEM_TEAM_WORLD, d_hits_total, d_hits, 1);
```

Listing 5: Launch kernel on the devices to perform the computations, block until all are finished, and then accumulate the results.

Estimating π

```
    if (my_pe == 0) {
        // Copy final result back to the host
        CUDA_CHECK(cudaMemcpy(hits, d_hits_total, sizeof(int),
↪   cudaMemcpyDeviceToHost));

        // Calculate final value of pi
        float pi_est = (float) *hits / (float) (N) * 4.0f;

        // Print out result
        std::cout << "Estimated value of pi averaged over all PEs = " << pi_est <<
↪   std::endl;
        std::cout << "Relative error averaged over all PEs = " << std::abs((M_PI -
↪   pi_est) / pi_est) << std::endl;
    }
```

Listing 6: Only on the first PE, copy accumlated hits back to host and report estimated value of π.

Estimating π

```
    free(hits);
    nvshmem_free(d_hits);
    nvshmem_free(d_hits_total);

    // Finalize nvshmem
    nvshmem_finalize();

    return 0;
}
```

Listing 7: Free memory on hosts and device, finalize NVSHMEM, and exit.

Estimating π

```bash
#!/bin/bash
#SBATCH -J nvshmem_pi         # Job name
#SBATCH -o nvshmem_pi_%j.out # Output file name
#SBATCH -p amd                # Queue (partition)
#SBATCH -c 1                  # Cores
#SBATCH --mem=6G              # Memory
#SBATCH --gres=gpu:1          # GPUs
#SBATCH -t 5                  # Time limit
```

Listing 8: Reguest compute resources.

Estimating π

```
echo $SLURM_JOB_PARTITION

module purge
module load nvhpc-22.2       # Alternatively nvhpc-21.2, nvhpc-21.9

NVSHMEM_HOME=/hpc/applications/nvidia/hpc_sdk/2022_22.2/Linux_x86_64/22.2/comm_libs/11.2/
GCC_HOME=/hpc/spack/opt/spack/linux-centos7-x86_64/gcc-7.3.0/gcc-9.2.0-
↪   6zgrndxveon2m5mjhltrqccdcewrdktx/bin
```

Listing 9: Setup the environment. `$GCC_HOME` is to provide full C++ 11 support.

Estimating π

```
nvcc -ccbin=$GCC_HOME -x cu -arch=sm_60 -rdc=true -I $NVSHMEM_HOME/include\
 -L $NVSHMEM_HOME/lib -lnvshmem -lcuda -o nvshmem_pi nvshmem_pi.cpp
# CUDA Compute Capabilities
# P100 sm_60
# V100 sm_70
# A100 sm_80

srun nvshmem_pi
```

Listing 10: Build the executable for the specific PE being used and run.