Riviera GPU Overview

Ben Blount

Data Science Research Institute

July 2025

COLORADO STATE UNIVERSITY

GPU Compute vs CPU Compute

- GPUs are much better at highly parallelized tasks and much worse at highly serialized tasks than CPUs due to the difference in the number of cores in the architectures of CPUs and GPUs.
  - CPUs have few very fast cores with advanced control units and access to much larger L1, L2, and L3 caches. [1]
  - GPUs have hundreds or even thousands of small slower cores that operate under a Single Instruction, Multiple Data (SIMD) model so a single control unit can control many cores at once. [1]
- GPUs in particular excel at tasks such as machine learning and scientific computing.

GPU vs CPU Diagram

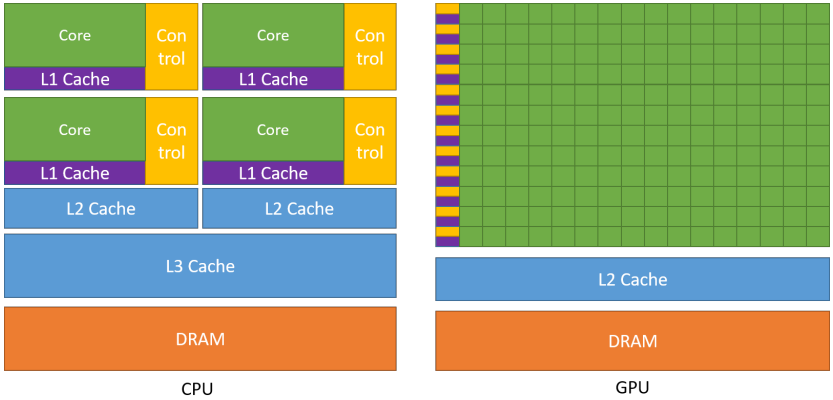

Figure: An example of CPU vs GPU core architecture from *CUDA C++ Programming Guide* [2]

What is CUDA (Compute Unified Device Architecture)

- CUDA is Nvidia's parallel computation platform and programming model that allows developers to do computation on CUDA supported GPUs.
- CUDA allows developers to interact directly with GPU hardware for parallelized tasks.
- CUDA works with a multitude of languages including C, C++, Fortran, Python, and Julia, but C++ and Python are the most common to use on Riviera.

Creating Kernels

Kernels are functions added to C++ by CUDA that get executed *N* times in parallel instead of a single time serially like normal functions in C++ are. To declare a Kernel a programmer uses the `__global__` declaration specifier.

```
__global__ void vector_add (float *out, float *a, float *b, int n) {
    for (int i = 0; i < n; i ++) {
        out[i] = a[i] + b[i];
    }
}
```

Calling Kernels

Once a kernel is declared, it can be called and assigned a number of threads using the `<<<a,b>>>` execution configuration syntax. When calling a kernel the first number `a` represents the number of blocks in the grid and the second number `b` represents the number of threads per block. This means that a total of $a \times b$ threads are spawned organized as `a` blocks of `b` threads.

```
int main()
{
        ...
        VecAdd<<1, N>>(A, B, C)
        ...
}
```

Memory Management

When using CUDA memory has to be specifically managed on the GPU. This is achieved by allocating memory, transferring data, and deallocating memory.

- In order to allocate memory use `cudaMalloc(void** devPtr, size_t size)` where `devPtr` is a pointer to a pointer that holds the device memory address and `size` is the number of bytes to allocate.
- When copying data use `cudaMemcpy` which copies data from the `src` pointer to the `dst` pointer using the protocol in `kind` (CPU $\rightarrow$ GPU, GPU $\rightarrow$ GPU, GPU $\rightarrow$ CPU).

```
cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind
↪  kind)
```

- In order to free memory use the `cudaFree(void* devPtr)`, which frees the memory at `devPtr`.

Memory Management Code

```
float *a, *b, *out, *d_a, *d_b, *d_out;
a = (float*)malloc(sizeof(float) * N);
b = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
... // Assign variables and do CPU related logic
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
vector_add<<<1,1>>>(d_out, d_a, d_b, N);
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_out);
free(a); free(b); free(out);
```

Sample SBatch Script

```
#!/bin/bash -l
#SBATCH --job-name=cuda-example
#SBATCH --partition=short-gpu
#SBATCH --output=out.log
#SBATCH --error=error.log
#SBATCH --time=00:01:00
#SBATCH --ntasks=1

module load cuda12.2/toolkit/12.2.2

srun nvcc vector_add.cu -o vector_add # Compiles the CUDA C++ program.
srun time ./vector_add # Runs and times the cuda program
```

Using PyTorch With CUDA

It is also possible to make use of CUDA through PyTorch. Here is a very basic PyTorch example to add two tensors together (much like our kernel in C++). Tensors are just abstractions of scalars, vectors, and matrices, where a scalar is a 0D tensor, a vector is a 1D tensor, and a matrix is a 2D tensor.

```python
import torch
device = torch.device(torch.accelerator.current_accelerator())
# Generates 2 1D PyTorch Tensors
a = torch.randn(10_000_000, device=device, dtype=torch.float32)
b = torch.randn(10_000_000, device=device, dtype=torch.float32)
c = a + b # The vector addition
print(f"PyTorch device: {device}")
print(f"Result shape: {c.shape}")
print(f"First five elements: {c[:5]}")
```

Sample SBatch Script

```
#!/bin/bash
#SBATCH --job-name=pytorch-cuda
#SBATCH --partition=short-gpu
#SBATCH --output=%A/out.out
#SBATCH --error=%A/err.err
#SBATCH --ntasks=1
#SBATCH --time=00:05:00

module load python39
module load cuda12.2/toolkit
source pytorch/bin/activate # Activate python virtual environment
srun python3 pytorch-cuda.py
deactivate # Deactivate python virtual environment with PyTorch installed
```

References

[1] "Cpu vs gpu: How they work and when to use them," datacamp. [Online]. Available: https://www.datacamp.com/blog/cpu-vs-gpu. (accessed: 07.22.2025).

[2] "Cuda c++ programming guide," Nvidia. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#. (accessed: 07.22.2025).