Parallel & Distributed Computing: Lecture 35

Alberto Paoluzzi

January 7, 2020

1. Introduction to CUDA

2. Julia packages for GPU programming

3. CuArrays.jl

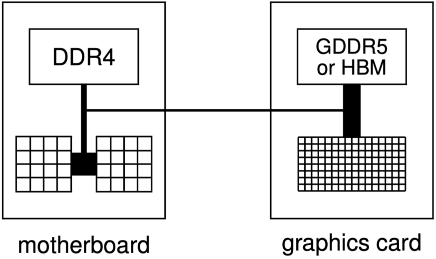
Introduction to CUDA

Hardware view

At the top-level:

- PCIe graphics card with a many-core GPU and
- high-speed graphics "device" memory

sits inside a standard PC/server with one or two multicore CPUs:



DGX-1 (Computational Sciences Lab – DMF)

The NVIDIA DGX-1 comes with a base operating system consisting of

- Ubuntu OS,
- Docker,
- Docker Engine Utility for NVIDIA GPUs,
- NVIDIA drivers.

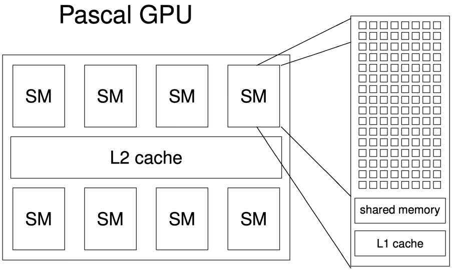
Ths system is designed to run a number of NVIDIA-optimized deep learning framework applications packaged in Docker containers.

DGX-1 Ip: tesla2.dia.uniroma3.it

- 2 Intel® Xeon®

E5-2698 v4, 20-core, 2.2GHz, 135W

- 8 Tesla P100 (Pascal architecture)
  - 170 teraflops, FP16
  - 16 GB memory per GPU
  - 28,672 NVIDIA CUDA® Cores



building block is a "streaming multiprocessor" (SM):

- 192 cores and 64k registers
- 64 KB of shared memory / L1 cache
- 8 KB cache for constants
- 48 KB texture cache for read-only arrays
- up to 2K threads per SM

Multithreading

cores in a SM are SIMT (Single Instruction Multiple Threads) cores:

- groups of 32 cores (warp) execute the same instructions simultaneously, but with different data
- 32 threads all doing the same thing at the same time
- natural for graphics processing and much scientific computing

SIMT is a natural choice for many-core chips to simplify each core

Software view

The master process runs on the CPU and performs the following steps:

1. initialises card
2. allocates memory in host and on device
3. copies data from host to device memory
4. launches multiple instances of execution "kernel" on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU:

1. each instance of the execution kernel executes on a SM
2. if # of kernel instances exceeds # SMs,
    - then more than one instance will run on each SM if there are enough registers and shared memory,
    - and the others will wait in a queue and execute later
3. all threads within one instance can access local shared memory but can't see what the other instances are doing (even if they are on the same SM)
4. there are no guarantees on the order in which the instances execute

CUDA programming

CUDA (Compute Unified Device Architecture) is NVIDIA's program development environment:

- based on C/C++ with some extensions
- FORTRAN support provided by compiler from PGI (owned by NVIDIA) and by IBM XL compiler
- lots of example code and good documentation
- large user community on NVIDIA forums

CUDA advantages

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

- Scattered reads – code can read from arbitrary addresses in memory
- Unified virtual memory (CUDA 4.0 and above)
- Unified memory (CUDA 6.0 and above)
- Shared memory – CUDA exposes a fast shared memory region that can be shared among threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

Cuda programming

Installing CUDA on a system, there are 3 components:

- driver
  - low-level software that controls the graphics card
- toolkit
  - nvcc CUDA compiler
  - Nsight IDE plugin for Eclipse or Visual Studio profiling and debugging tools
  - several libraries
- SDK
  - lots of demonstration examples
  - some error-checking utilities
  - not officially supported by NVIDIA almost no documentation

Cuda programming (from https://en.wikipedia.org/wiki/CUDA)

CUDA 8.0 comes with the following libraries (in alphabetical order):

- cuBLAS – CUDA Basic Linear Algebra Subroutines library, see main and docs
- CUDART – CUDA Runtime library, see docs
- cuFFT – CUDA Fast Fourier Transform library, see main and docs
- cuRAND – CUDA Random Number Generation library, see main and docs
- cuSOLVER – CUDA based collection of dense and sparse direct solvers, see main and docs
- cuSPARSE – CUDA Sparse Matrix library, see main and docs
- NPP – NVIDIA Performance Primitives library, see main and docs
- nvGRAPH – NVIDIA Graph Analytics library, see main and docs
- NVML – NVIDIA Management Library, see main and docs
- NVRTC – NVIDIA Runtime Compilation library for CUDA C++, see docs

Tutorial 01: Say Hello to CUDA (C language)

CUDA Tutorial

Search docs

Home
Tutorials
Tutorial 01: Say Hello to CUDA
Tutorial 01: Say Hello to CUDA
Introduction
A quick comparison between CUDA and C
Compiling CUDA programs
Putting things in actions.
Exercise: Converting vector addition to CUDA
Profiling performance
Wrap up
Acknowledgments
Tutorial 02: CUDA in Actions

« Previous Next »

Docs » Tutorials » Tutorial 01: Say Hello to CUDA

Tutorial 01: Say Hello to CUDA

Introduction

This tutorial is an introduction for writing your first CUDA C program and offload computation to a GPU. We will use CUDA runtime API throughout this tutorial.

CUDA is a platform and programming model for CUDA-enabled GPUs. The platform exposes GPUs for general purpose computing. CUDA provides C/C++ language extension and APIs for programming and managing GPUs.

In CUDA programming, both CPUs and GPUs are used for computing. Typically, we refer to CPU and GPU system as *host* and *device*, respectively. CPUs and GPUs are separated platforms with their own memory space. Typically, we run serial workload on CPU and offload parallel computation to GPUs.

A quick comparison between CUDA and C

Following table compares a hello world program in C and CUDA side-by-side.

C

```
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
```

CUDA

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
```

v: latest

Julia packages for GPU programming

https://devblogs.nvidia.com/gpu-computing-julia-programming-language/

Julia is already well regarded for programming multicore CPUs and large parallel computing systems,

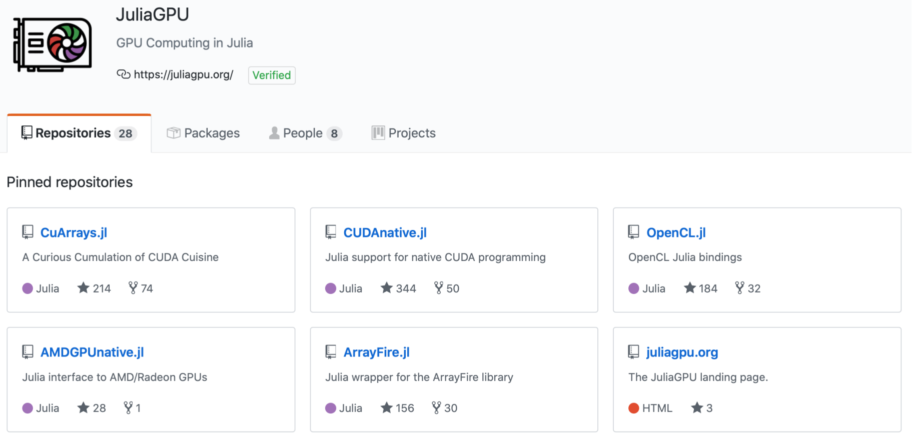
recent developments make the language suited for GPU computing as well.

The performance possibilities of GPUs can be democratized by providing more high-level tools that are easy to use by a large community of applied mathematicians and machine learning programmers.

In this blog post, I will focus on native GPU programming with a Julia package that enhances the Julia compiler with native PTX code generation capabilities: `CUDAnative.jl`

High-Performance GPU Computing in the Julia Programming Language

JuliaGPU — GPU Computing in Julia



JuliaGPU — CUDA programming in Julia

CUDA programming in Julia

This repository hosts a Julia package that bundles functionality from several other packages for CUDA programming, and provides high-level documentation and tutorials for effectively using CUDA GPUs from Julia. The documentation is accessible at juliagpu.gitlab.io.

CUDA.jl includes functionality from the following packages:

1. CUDAdrv.jl: interface to the CUDA driver
2. CUDAnative.jl: kernel programming capabilities
3. CuArrays.jl: GPU array abstraction

CuArrays.jl

API reference documentation of CuArrays.jl. This documentation is a work in progress. For general usage instructions of CuArrays.jl and the rest of the Julia CUDA toolchain, please refer to the CUDA.jl documentation.

CuArrays.jl

CuArrays.jl

CuArrays.jl provides a fully-functional GPU array, which can give significant speedups over normal arrays without code changes. CuArrays are implemented fully in Julia, making the implementation elegant and extremely generic

Read chapter 7 of book:

Avik Sengupta, Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond, 2nd Edition, Pakt>, 2019

Getting started with GPUs

CUDA, PTX and its use from Julia

CuArrays - Arrays on the GPU

Analyzing the performance of GPU code