Introduction to GPU Programming

Wim R.M. Cardoen

Center of High-Performance Computing (CHPC)
University of Utah

October 14, 2024

Outline I

Outline II

- Protected env.: redwood cluster
- How to access the GPUs at CHPC

5 Questions

Motivation

Theoretical GFLOP/s: GPU vs. CPU

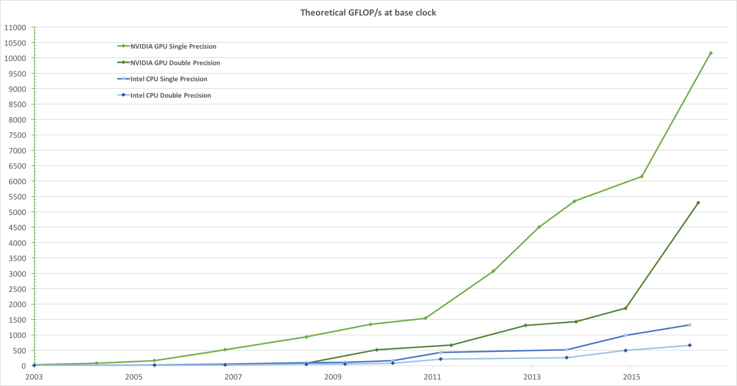
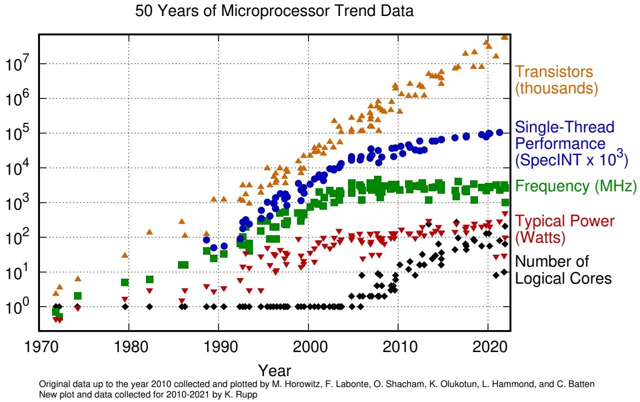


Figure: Theoretical GFLOP/s: GPUs vs. CPUs.[a]

[a] https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf

CPU processor trend (last 50 years)



- After the year 2000, `freq./power` for a single CPU core reaches a max. (**Heat dissipation!**).

Energy efficiency per job: GPU vs. CPU

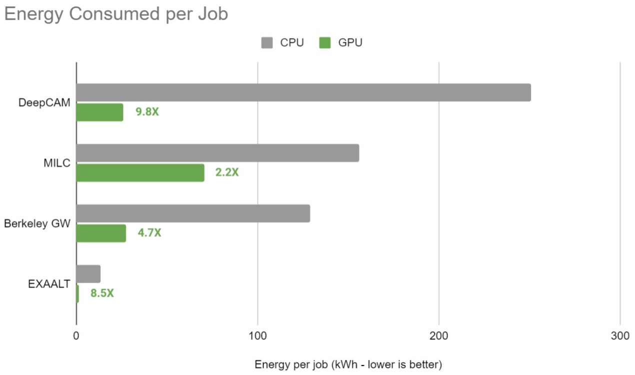


Figure: Energy efficiency per job (NERSC).[a]

[a] https://blogs.nvidia.com/blog/gpu-energy-efficiency-nersc/ (05/21/2023)

Hardware

Streaming Multiprocessor (SM)

- GPU device connected to the CPU by a PCIe bus.
- each GPU device contains an array (**x**) of Streaming Multiprocessors **(SM)**.
- each SM has:
  - a Single-Instruction Multiple-Thread (`SIMT`) Architecture.
  - contains **y** regular cores and [**z** tensor cores].
- scalable: newer generations: increase of **x**, **y** and [**z**], e.g.:
  - `NVIDIA A100-PCIE-40GB` (*notch293*)
    - global memory: 40 GB.
    - 108 SMs, 64 Cores/SM, 4 Tensor Cores/SM.
    - GPU Max. Clock Rate: 1.41 GHz.
  - `NVIDIA H100 SXM5 NVL` (*grn008*)
    - global memory: 93 GB.
    - 132 SMs, 128 Cores/SM, 4 Tensor Cores/SM.
    - GPU Max. Clock Rate: 1.78 GHz.

NVIDIA GH100 SM

Figure: GH100 SM

NVIDIA GH100 Full Device

Figure: NVIDIA GH100 Full Device (144 SMs).

GPU Threads - Warps

- Each SM:
  - generates, schedules, executes threads in batches of 32 threads.
  - **WARP**: a batch of 32 threads
- each thread in a WARP executes the same instructions but runs its own "path".
- if threads within a WARP diverge, the threads become inactive/disabled.

Software

GPGPU & CUDA

- GPU (Graphic Processing Unit):
  orginally developed for graphical applications.
- GP-GPU: General-Purpose GPU, i.e.
  the use of GPUs beyond graphical applications.
  CAVEAT: problem to be reformulated in terms of the graphics API.
- **2007**: NVIDIA introduces the **CUDA**[1] framework
  (**C**ompute **U**nified **D**evice **A**rchitecture)
  - CUDA API: extension of the `C` language.
  - handles the GPU thread level parallelism.
  - deals with moving data between CPU and GPU.
  - also support for `C++`, `Fortran` and `Python`.

[1] The CUDA Toolkit consists of 2 parts:

- CUDA Driver
- CUDA Toolkit (`nvcc`,`nvprof`, . . . , libraries, header files).

Schema of CUDA Components

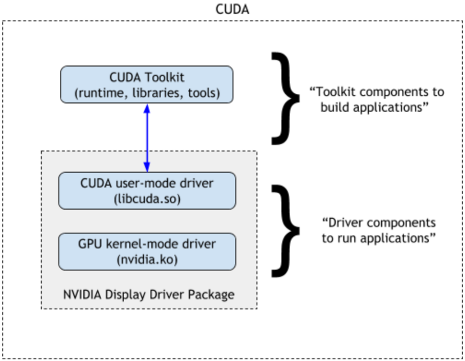


Figure: Schema of the CUDA Components

Structure of a GPU computation

1. **Allocate** memory space on the GPU device.
2. **Transfer** the data from the CPU to the GPU device.
3. Perform the **calculation** on the GPU device.
    - **kernel**: function executed on the GPU.
    - To enhance performance: keep data as long as possible on the GPU device.
4. **Transfer** the result back from the GPU device to the CPU.
5. **Deallocate** memory space on the GPU device.

Note: source code & makefile available in `./src`

Alloc. & free of global memory on the GPU

- `cudaError_t`
  CUDA Error types.
- `cudaError_t cudaMalloc(void **devPtr, size_t size)`
  Allocates memory on the device.
- `cudaError_t cudaFree(void *devPtr)`
  Frees memory on the device.

```c
double *M_d, *N_d, *P_d;  // Pointers (device)
int const SZ=16;
int const SZ2=SZ*SZ;

// Allocate M on device (M_d)
if(cudaMalloc(&M_d,sizeof(double)*SZ2) != cudaSuccess)
{
   printf(" ERROR: alloc vector M on DEVICE \n");
   return 1;
}

// Deallocate matrix M_d
if(cudaFree(M_d) != cudaSuccess)
{
   printf(" ERROR: unable to deallocate M_d (DEVICE)\n");
   return 1;
}
```

Listing 1: `Alloc/Free extract`

Copy data between host (CPU) and device (GPU)

- Copy data bewteen host (CPU) and device (CPU)

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

- Direction (`kind`):
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

```c
// Copy M from Host onto Device
if(cudaMemcpy(M_d, M_h, sizeof(double)*SZ2,
              cudaMemcpyHostToDevice) != cudaSuccess)
{
   printf("ERROR: copy vector M: HOST -> DEVICE\n");
   return 1;
}


// Copy P (result) from the Device to the Host
if(cudaMemcpy(P_h, P_d, sizeof(double)*SZ2,
              cudaMemcpyDeviceToHost) != cudaSuccess)
{
   printf("ERROR: copy vector N: DEVICE -> HOST\n");
   return 1;
}
```

Listing 2: `cudaMemcpy extract`

CUDA Kernel

- **CUDA kernel**: alias for a function which may run on a GPU device.
- **Kernel declaration** syntax:

  funcspec `void` *kernelName*(*args*){ *body* }

  where:
  - *funcspec*: function type qualifier, i.e. `__global__`,`__host__`,`__device__`
  - *kernelName*: name of the kernel/CUDA function.
  - *args*: argument list of the kernel/CUDA function.
  - *body*: body of the kernel/CUDA function (your code).
- **Kernel call** syntax:

  kernelName<<<*gridSize,blockSize*>>>(*args*)

  where:
  - *gridSize*: size of the grid of thread blocks.
  - *blockSize*: size of a thread block.

Function type qualifiers

Qualifier	Called from	Executed on
`__global__`	host	device
`__host__`	host	host
`__device__`	device	device

Table: Function type qualifiers

Note:

- You can have to different versions of a function i.e.: one with `__host__` & one with `__device__`

Grids, Blocks and Threads

We have a hierarchical (software) implementation.

- `uint3`,`dim3`:
  - CUDA defined structures of unsigned integer `x,y,z`
  - `dim3`: based on `uint3` but unspecified components are initialized to 1.
- **Grid**: each Grid consists of Blocks
  - `dim3 gridDim`: dimensions of the Grid.
  - `uint3 blockIdx`: block index within the Grid.
- **Block**: each Block consists of Threads
  - `dim3 blockDim`: dimensions of the Block
  - `uint3 threadIdx`: thread index within the block.

Preliminary notion: matrix storage as a 1D vector

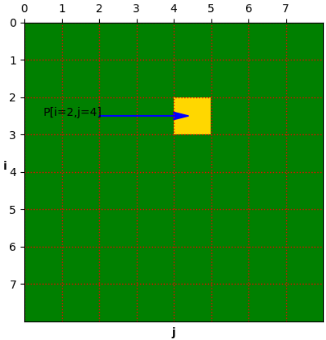


Figure: P: 8x8 Matrix ($d = 8$)

- $P[i,j]$ is stored as 1D vector: $P[i * d + j]$ where $i, j \in [0, 8)$.

Matrix Multiplication $P = M\ N$

Matrix multiplication: $P = M\ N$ where $M, N\ \in\ \mathbb{R}^{dxd}$

- $P[i,j] = \sum_{k=0}^{d-1} M[i,k]\ N[k,j]$
- $M[i,k]$ is stored as: $M[i*d+k]$
- $N[k,j]$ is stored as: $N[k*d+j]$
- Therefore,

$$P[i*d+j] = \sum_{k=0}^{d-1} M[i*d+k]\ N[k*d+j] \tag{1}$$

Matrix Mul.: kernel (v. 1)

- Each Thread (threaIdx) is represented as a 2D object, i.e. (`threadIdx.x`, `threadIdx.y`) (cfr. a point in plane geometry)
- Each Thread calculates only 1 element of P.

Implementation of Eq. (1) using 1 Block of Threads.

```
#include <mul.h>

__global__ void MatrixMulKernel1(double *M_d, double *N_d,
                                 double *P_d, int const SZ)
{
    double Pval=0;
    int tx=threadIdx.x;
    int ty=threadIdx.y;
    for(int k=0; k<SZ; k++)
        Pval+=M_d[tx*SZ +k]*N_d[k*SZ+ty];
    P_d[tx*SZ+ty]=Pval;
    return;
}
```

Listing 3: `Kernel (v. 1)`

Invoking kernel (v. 1)

- Invoking 1 Block of Threads

```
int main(void)
{
    int const SZ=16;
    // ...
    // Invoke Kernel to generate P=MxN
    dim3 dimBlock(SZ,SZ,1);
    dim3 dimGrid(1,1,1);
    MatrixMulKernel1<<<dimGrid,dimBlock>>>(M_d,N_d,P_d,SZ);
    //..
}
```

Listing 4: `Invoking kernel (v. 1)`

Mat. Mul (v.2): Grid of 2D Blocks

- `int tx = blockIdx.x*blockDim.x + threadIdx.x;`
- `int ty = blockIdx.y*blockDim.y + threadIdx.y;`

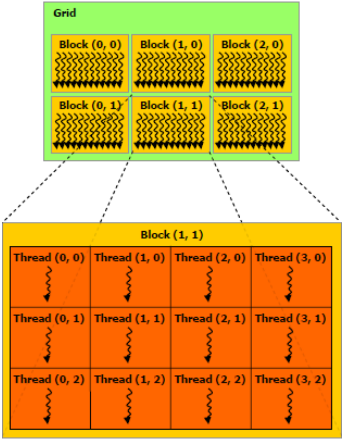


Figure: 2D-Grid of 2D-Blocks of Threads

Matrix Mul.: visualization (v. 2)

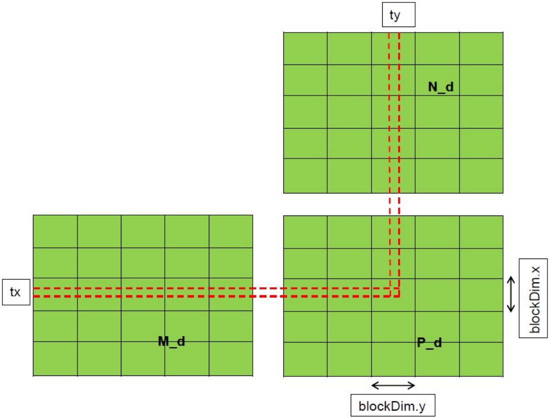


Figure: Matrix Mul. (2D Grid)

Matrix Mul.: kernel (v. 2)

```
#include <mul.h>

__global__ void MatrixMulKernel2(double *M_d, double *N_d,
                                 double *P_d, int const SZ)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

    if((tx<SZ) && (ty<SZ))
    {
       double Pval=0.0;
       for(int k=0; k<SZ; k++)
           Pval += M_d[tx*SZ+k]*N_d[k*SZ+ty];
       P_d[tx*SZ+ty]=Pval;
    }
    return;
}
```

Listing 5: `Kernel (v. 2)`

Invoking kernel (v. 2)

- Invoking a grid of blocks of threads

```
int main(void)
{
    int const SZ=500;

    // ..
    int const THREADX=16;
    int const THREADY=16;
    dim3 dimBlock(THREADX,THREADY,1);
    int numBlocksX=(SZ%THREADX==0 ? SZ/THREADX : SZ/THREADX +1);
    int numBlocksY=(SZ%THREADY==0 ? SZ/THREADY : SZ/THREADY +1);
    dim3 dimGrid(numBlocksX,numBlocksY,1);

    MatrixMulKernel2<<<dimGrid,dimBlock>>>(M_d,N_d,P_d,SZ);
    //..
}
```

Listing 6: `Invoking kernel (v. 2)`

Types of GPU memory

- **global memory**: (largest, slowest and often the bottleneck).
- **constant memory**: cached, read-only
  - `__constant__`: constant memory space specifier
- **registers**: fast, on-chip memory (exclusive to each thread).
- **shared memory**: allocated per thread block & low latency
  - `__shared__`: shared memory space specifier
  - `__syncthreads()`: barrier function which forces all threads in a block to wait until all threads have arrived before proceeding. (block level synchronization).

Matrix Mul.: use of shared memory (v. 3)

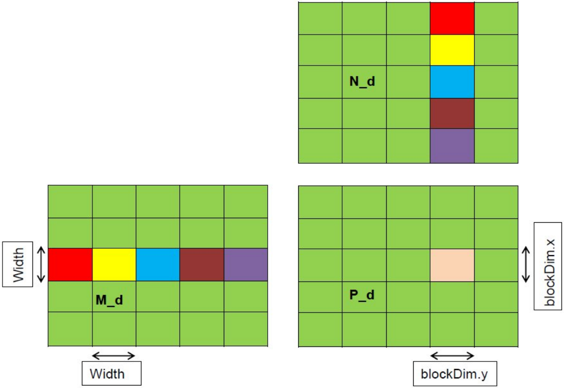


Figure: Matrix Mul.: use of shared memory

Matrix Mul.: kernel (v. 3) - use of shared memory

```
#include <mul.h>
#define WIDTH 16
__global__ void MatrixMulKernel3(double *M_d, double *N_d, double *P_d, int const SZ)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

    __shared__ double M_s[WIDTH][WIDTH];
    __shared__ double N_s[WIDTH][WIDTH];

    double Pval=0.0;
    int nslices=(SZ%WIDTH==0)?(SZ/WIDTH):(SZ/WIDTH+1);
    for(int islice=0; islice<nslices; islice++)
    {
        M_s[threadIdx.x][threadIdx.y]=M_d[tx*SZ + islice*WIDTH + threadIdx.y];
        N_s[threadIdx.x][threadIdx.y]=N_d[islice*WIDTH*SZ + threadIdx.x*SZ +ty ];
        __syncthreads();

        for(int k=0; k<WIDTH; k++)
            Pval += M_s[threadIdx.x][k]*N_s[k][threadIdx.y];
        __syncthreads();
    }

    if(tx<SZ && ty<SZ)
       P_d[tx*SZ+ty]=Pval;
    return;
}
```

Listing 7: `Kernel (v.3)`

Building/Compiling CUDA applications

General scheme:

- Source code for CUDA applications:
  - `C/C++` host code with extensions to deal with the device(s).
  - Other programming languages are allowed e.g. `Fortran`
- Primo: **separate** device functions from host code.
- **Device code**: preprocessing, compilation with the NVIDIA compiler (`nvcc`).
- **Host code**: preprocessed, compiled with a host (`C/C++`) compiler e. g. (`gcc, g++, icc, icpc, ...`)
- Compiled device functions are **embedded** as `fatbinary` images in the host object file.
- **Linking stage**: adding CUDA runtime libraries to the host object file to create an executable.

Further concepts

- `.cu` : Suffix for CUDA source file (host code (`C`,`C++`) & device code).
- `.cuf`: Suffix for CUDA source file (host code (`Fortan`) & device code).
- `.ptx`: Suffix for **P**arallel **T**hread E**x**ecution (`PTX`) files. An intermediate representation (similar to assembly for a **virtual GPU architecture**[2]
- `.cubin`: Suffix for the **CU**DA device **bin**ary file pertaining to a **real GPU architecture**[3]
- `fatbin`: Multiple `PTX` [& `cubin`] files are merged into a `fatbin` file.

[2] Virtual architectures bear the "`compute_`" **prefix** e. g. "`compute_70`".
[3] Real (physical) architectures bear the "`sm_`" **prefix** e. g. "`sm_70`".
Memento: "`sm`" stands for the physical streaming multiprocessor.

Compilation trajectory (cont.)

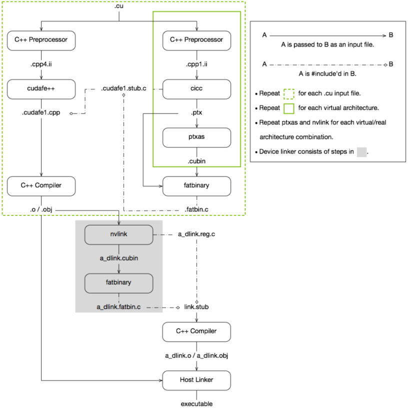


Figure: Compilation trajectory

In praxi,

We will now address the following questions:

- What are the recent CUDA architectures
- How to find the compute capability (CC) of a device
- How to build an executable for a particular device.
- How to build an executable for multiple architectures

Recent CUDA Architectures/Generations

- NVIDA GPU: *xy* *x*:generation (major) *y*: minor
- new generation: major improvement in functionality/chip
- **binary** compatibility is **NOT** garantueed among generations.

Architecture/ Generation	Year	compute_ (*virtual*)	sm_ (*real*)
Maxwell	2014	50, 52, 53	50, 52, 53
Pascal	2016	60, 61, 62	60, 61, 62
Volta	2017	70, 72	70, 72
Turing	2018	75	75
Ampere	2020	80, 86, 87	80, 86, 87
Ada Lovelace	2022	89	89
Hopper	2022	90, 90a	90, 90a

Table: Some of the recent CUDA architectures (10/08/2024)

Retrieval of the Compute Compability (CC)

You can:

1. use the cmd `nvidia-smi` to display the architecture.
   The cmd `nvidia-smi` is **extremely powerful** tool to query the state & specs of the devices attached to a particular node.[4]
2. use the cmd `nvaccelinfo` (part of NVIDIA HPC SDK)
3. write some basic C/C++ code [5]. relying on the following CUDA APIs:
   - `cudaGetDeviceCount(int *tot)`:
     returns the number of devices on the localhost.
   - `cudaGetDeviceProperties(cudaDeviceProp *p, int idev)`:
     returns information about the compute-device `idev`.

[4] **Note**: it is recommended to investigate the different flags pertaining to the cmd.
[5] Already available in `src/devicequery`.
The name of the corresponding executable is `devinfo`

Use of nvidia-smi

```
[u0253283@notch001:~]$ nvidia-smi -l
Tue Oct 22 12:38:52 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.14              Driver Version: 550.54.14      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla V100-PCIE-16GB           On  |   00000000:3B:00.0 Off |                  Off |
| N/A   31C    P0             24W /  250W |       0MiB /  16384MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
^C[u0253283@notch001:~]$ nvidia-smi --query-gpu=compute_cap --format=csv
compute_cap
7.0
```

Figure: nvidia-smi

Retrieval of CC using nvaccelinfo

```
[u0253283@notch001:~]$ module load nvhpc
[u0253283@notch001:~]$ nvaccelinfo

CUDA Driver Version:           12040
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  550.54.14  Thu Feb 22 01:44:30 UTC 2024

Device Number:                 0
Device Name:                   Tesla V100-PCIE-16GB
Device Revision Number:        7.0
Global Memory Size:            16928342016
Number of Multiprocessors:     80
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1380 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             877 MHz
Memory Bus Width:              4096 bits
L2 Cache Size:                 6291456 bytes
Max Threads Per SMP:           2048
Async Engines:                 7
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
  Multi-Device:                Yes
Default Target:                cc70

Device Number:                 1
Device Name:                   Tesla V100-PCIE-16GB
```

Figure: Use of nvaccelinfo to retrieve compute_

Retrieval of CC through some simple CUDA APIs

```
[u0253283@notch001:~]$ devinfo
Node: notch001
#Devices detected: 3


  Device:0
    Device Name : Tesla V100-PCIE-16GB
    ECC Enabled            : 0
    Compute Capability     : 70
    Compute Mode           : 0
    #SM on device          : 80
    Device Clock Rate (GHz) : 1.3800E+00
    Peak Memory clock frequency (GHz): 8.7700E-01
    L2 Cache Size (bytes)           : 6291456
    Warp Size in Threads            : 32

    Max. #Blocks/SM                 : 32
    Max. Size of each dim. of a Grid : (2147483647,65535,65535)
    Max. Size of each dim. of a Block: (1024,1024,64)
    Max. #Threads/Block             : 1024
    Max. Resident Threads/SM        : 2048
    Global Mem. available on device (bytes)  : 16928342016
    Constant Mem. available on device (bytes): 65536
    #32-bit Registers available per SM       : 65536
    #32-bit Registers available per Block    : 65536
    Shared Mem. available per Block (bytes)  : 49152
    Shared Mem. available per SM (bytes)     : 98304


  Device:1
    Device Name : Tesla V100-PCIE-16GB
    ECC Enabled            : 0
    Compute Capability     : 70
```

Figure: `devinfo` based on a few CUDA APIs

Compiling your code for a particular device/devices

- Compilation goes in 2 steps:
  1. `PTX representation`: generic assembly instructions for a *virtual* (`compute_` prefix) GPU architecture.
     The resulting `.ptx` file is human readable (**text** file).
  2. `Binary generation`: generation of an **object** file for the *real* (`sm_` prefix) GPU architecture (based on the `PTX` file).
- `-arch`/`-code` flags
  - `--gpu-architecture|-arch <arch>`: specifies the name of 1 *virtual* GPU architecture for which the code needs to be compiled.
    e. g. `-arch=compute_50`
  - `--gpu-code|-code <arch>`: specifies the name(s) of the *real* GPU architecture(s) for which the binary needs to be compiled.
    e. g. `-code=sm_52`

Compiling your code (cont.)

- Therefore, choose 1 virtual architecture and the accompanying real architectures

  e. g. `-arch=compute_50 -code=sm_50,sm_51,sm_52`
  - `PTX` file generated for the `compute_50` (*virtual*) arch.
  - **fatbinary** object created for the (`real`) arch. `sm_50,sm_51,sm_52`
- `--generate-code|-gencode arch=<arch>,code=<code> ...`
  - **Generalization** of the previous construct:
    `--gpu-architecture=<arch> --gpu-code=<code>`.
  - allows the creation of binaries for different architectures.
  - example:

```
# Compiler flags simultaneously targeting
# Maxwell & Pascal architectures
-gencode arch=compute_50,code=sm_50 \
-gencode arch=compute_52,code=sm_52 \
-gencode arch=compute_53,code=sm_53 \
-gencode arch=compute_60,code=sm_60 \
-gencode arch=compute_61,code=sm_61 \
-gencode arch=compute_62,code=sm_62
```

Profiling & debugging

CUDA SDK comes with:

- its own profiler: `nvprof`.
- its own debugger: `nvsight`

Profiling mul3 using nvprof

```
[u0253283@notch001:3]$ nvprof ./mul3
==2424062== NVPROF is profiling process 2424062, command: ./mul3
 Calling Kernel ...
 Kernel Call Finished ...

 Frob. Norm(P-P_h):    0.0000000000

==2424062== Profiling application: ./mul3
==2424062== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   59.70%  1.1707ms         1  1.1707ms  1.1707ms  1.1707ms  MatrixMulKernel3(double*, double*, double*, int)
                   32.02%  627.87us         2  313.93us  307.13us  320.73us  [CUDA memcpy HtoD]
                    8.28%  162.33us         1  162.33us  162.33us  162.33us  [CUDA memcpy DtoH]
      API calls:   95.82%  138.45ms         3  46.151ms  109.78us  138.23ms  cudaMalloc
                    2.88%  4.1668ms         3  1.3889ms  494.15us  3.1420ms  cudaMemcpy
                    0.82%  1.1901ms         1  1.1901ms  1.1901ms  1.1901ms  cudaLaunchKernel
                    0.27%  395.44us         3  131.81us  94.455us  202.38us  cudaFree
                    0.17%  250.22us       114  2.1940us     291ns  103.89us  cuDeviceGetAttribute
                    0.01%  16.428us         1  16.428us  16.428us  16.428us  cuDeviceGetPCIBusId
                    0.01%  12.713us         1  12.713us  12.713us  12.713us  cuDeviceGetName
                    0.00%  1.5660us         3     522ns     321ns     905ns  cuDeviceGetCount
                    0.00%  1.2730us         2     636ns     338ns     935ns  cuDeviceGet
                    0.00%     613ns         1     613ns     613ns     613ns  cuModuleGetLoadingMode
                    0.00%     556ns         1     556ns     556ns     556ns  cuDeviceTotalMem
                    0.00%     547ns         1     547ns     547ns     547ns  cudaGetLastError
                    0.00%     337ns         1     337ns     337ns     337ns  cuDeviceGetUuid
```

Figure: Profiling mul3 on notch001

Important CUDA libraries

In order to increase the performance of your code we recommend to use highly-optimized libraries. Among them, we have:

- **cuBLAS**: **B**asic **L**inear **A**lgebra **S**ubroutines on NVIDIA GPUs.
- **MAGMA**: **M**atrix **A**lgebra on **G**PU and **M**ulti-core **A**rchitectures.
- **cuRAND**: Random Number Generation library.
- **cuFFT**: CUDA **F**ast **F**ourier **T**ransform library.
- **NCCL**: **N**VIDIA **C**ollective **C**ommunications **L**ibrary.
- **cuDNN**: CUDA **D**eep **N**eural **N**etwork library.
- **cuTENSOR**: GPU-accelerated Tensor Linear Algebra.
- **DALI**: NVIDIA **Da**ta Loading **Li**brary.
- ...

Alternatives to CUDA

- Similar to CUDA
  - **ROCM** (AMD)
- **OpenACC** (use of directives (cfr. `OpenMP`)
  - GCC: supports OpenACC for NVIDIA & AMD GPUs.
  - NVIDIA HPC SDK (formerly PGI)
  - Sourcery Codebench (AMD GPU)
- Higher-level abstractions
  - **Kokkos** (prog. model for parallel algorithms for many-core chips)

Links

- CUDA Toolkit Documentation
- CUDA C++ Programming Guide Release 12.6 (10/01/24)
- CUDA C++ Best Practices Guide, Release 12.6 (09/24/24)
- NVIDIA CUDA Compiler Driver NVCC, Release 12.6 (09/24/24)
- PTX & ISA Release 8.5 (09/24/24)

Use of GPUs at the CHPC

GPU devices on lp/kp/np/grn

GPU device type	compute capability
NVIDIA GeForce GTX TITAN X	5.2
Tesla P100-PCIE-16GB	6.0
Tesla P40	6.1
NVIDIA GeForce GTX 1080 Ti	6.1
NVIDIA Titan V	7.0
NVIDIA Tesla V100-PCIE-16GB	7.0
Tesla T4	7.5
NVIDIA GeForce RTX 2080 Ti	7.5
NVIDIA A100-PCIe-40GB	8.0
NVIDIA A100-SXM4-80GB	8.0
NVIDIA A800 40GB Active	8.0

Table: GPU devices on lp/kp/np/grn (10/01/2024)

GPU devices on lp/kp/np/grn (cont.)

GPU device type	compute capability
NVIDIA GeForce RTX 3090	8.6
NVIDIA A40	8.6
NVIDIA RTX A5500	8.6
NVIDIA RTX A6000	8.6
NVIDIA RTX 6000 Ada Generation	8.9
NVIDIA L40	8.9
NVIDIA L40S	8.9
NVIDIA H100 NVL/Deep Dive	9.0

Table: GPU devices on lp/kp/np/grn (10/01/2024)

GPU devices on redwood

GPU device type	compute capability
NVIDIA GeForce GTX 1080 Ti	6.1
NVIDIA A100-SXM4-40GB	8.0
NVIDIA A100 80GB PCIe	8.0
NVIDIA A30	8.0
NVIDIA A40	8.6
NVIDIA RTX 6000 Ada Generation	8.9
NVIDIA H100 NVL/Deep Dive	9.0

Table: GPU devices on redwood (10/01/2024)

Accessing GPUs at CHPC

- Using GPUs at the CHPC (Presentation by Martin Čuma)
- **Note:**
  - When a GPU job is launched the job runs with its own cgroup. (limits/accounts for its own resources).
  - When a $USER has several GPU jobs running on the **same** node, the $USER will land in **one** cgroup belonging to one of his/her jobs when the $USER `ssh`es into the node where these jobs run.
    Therefore, the $USER can **not** verify the status of his/her other jobs using tools like `nvidia-smi` directly.
  - ⇒ use,
    `srun --pty --overlap --jobid $JOBID /usr/bin/nvidia-smi`
    where JOBID stands for the jobid.

Questions

Questions?

Thank you!
Any questions?