Introducing GPUs Architecture - Important Aspects -

Carlos Jaime BARRIOS HERNANDEZ, PhD.
## About Top500 List - 2022

### Top 5

1. **Frontier** - HPE Cray EX230, AMD Optimized 3rd Generation EPYC 7642 2.6GHz, NVDIA A100, HPE Supercomputing Systems
2. **Supercomputer Fugaku** - Supercomputer Fugaku, A44X 4X 2.20GHz, Taihu interconnect O, Fujitsu
3. **LUMI** - HPE Cray Ex230, AMD Optimized 3rd Generation EPYC 7642 2.6GHz, NVDIA A100, HPE Supercomputing Systems
4. **Leonardo** - BullSequana X10000, Xeon Platinum 8390 2.40GHz, NVDIA A100/SXM4 4x GB, Quad-rail NVDIA HDR100 Infiniband, Atos
5. **Summit** - IBM Power System AC922, IBM Power9 22C 3.07GHz, NVDIA Volta V100, Dual-rail Mellanox EDR Infiniband, IBM

### Top 6-10

6. **Sierra** - IBM Power System AC922, IBM Power9 22C 3.10GHz, NVDIA Volta V100, Dual-rail Mellanox EDR Infiniband, IBM / NVDIA / Mellanox
7. **Sunway TaihuLight** - Sunway MP2, Sunway SW26010 288C 2.45GHz, Sunway, NCSPC
8. **Perlmutter** - HPE Cray EX230, AMD EPYC 7763 64C 2.65GHz, NVDIA A100/SXM4 4x GB, Slierhugger-10, HPE
9. **Selene** - NVIDIA DGX A100, AMD EPYC 7762 64C 2.25GHz, NVDIA A100, Mellanox HDR Infiniband, NVIDIA Corporation
10. **Tianhe-2A** - Ti-HB-FEP Cluster, Intel Xeon E5-2697v2 2.20GHz, Ti Express-2, Mahr-2000, NJIT

### More Information

- [www.top500.org](http://www.top500.org)
- Top500 List - 2022
- Rankings and data updated on [www.top500.org](http://www.top500.org)

---

**Ranks 33-40**

33. **Pégaso** - Supermicro A+ Server 412600-N4RT+, AMD EPYC 7543 32C 2.60GHz, NVDIA A100, Infiniband HDR, Atos
34. **Dragão** - Supermicro SYS-4229GP-TVR7, Xeon Gold 6230R 26C 2.10GHz, NVDIA Tesla V100, Infiniband EDR, Atos
35. **Atlas** - Bull 4296P-TVR7, Xeon Gold 6340 18C 2.60GHz, NVDIA Tesla V100, Infiniband EDR, Atos
36. **IARA** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVDIA A100 SXM4 4x GB, Infiniband, NVIDIA
37. **NOB21** - ThinkSystem C239F, Xeon Platinum 8280 28C 2.70GHz, Broadcom, Lenovo
38. **A16A** - ThinkSystem D364, Xeon Gold 6252 24C 2.10GHz, 100G Ethernet, Lenovo
39. **Santos Dumont (S Dumont)** - Bull Sequana X1000, Xeon Gold 6252 24C 2.10GHz, Mellanox Infiniband EDR, NVDIA Tesla V100 SXM4, Atos

---

**Power**

- Power (kW)
- [www.top500.org](http://www.top500.org)
Architecture Terminology

- **Host** The CPU and its memory (host memory)
- **Device** The GPU and its memory (device memory)
GPGPU Accelerate Computing

Latency Processor + Throughput processor
1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Anatomy of a CUDA Application

Serial code executes in a **Host** (CPU) thread.

Parallel code executes in many **Device** (GPU) threads across multiple processing elements.
CUDA Kernels

Parallel portion of application: execute as a kernel
Entire GPU executes kernel, many threads

CUDA threads:
Lightweight
Fast switching
1000s execute simultaneously

<table>
<thead>
<tr>
<th>CPU</th>
<th>Host</th>
<th>Executes functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device</td>
<td>Executes kernels</td>
</tr>
</tbody>
</table>
CUDA Kernels: Parallel Threads

A **kernel** is a function executed on the GPU as an array of threads in parallel.

All threads execute the same code, can take different paths.

Each thread has an ID:
- Select input/output data
- Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
Blocks are grouped into a grid
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
Blocks are grouped into a grid
A kernel is executed as a grid of blocks of threads
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
Blocks are grouped into a grid
A kernel is executed as a grid of blocks of threads
Kernel Execution

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

Threads may need to cooperate:

- Cooperatively load/store blocks of memory that they all use
- Share results with each other or cooperate to produce a single result
- Synchronize with each other
Thread blocks allow scalability

Blocks can execute in any order, concurrently or sequentially.
This independence between blocks gives scalability:

A kernel scales across any number of SMs.
Memory hierarchy

Thread:
- Registers
- Local memory

Block of threads:
- Shared memory

All blocks:
- Global memory
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);

int N = 1<<20;
std::vector<float> x(N), y(N);
...

// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);

int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(), d_y.begin(),
                  2.0f * _1 + _2);
CUDA Fortran

module mymodule contains
    subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        do i=1,n
            y(i) = a*x(i)+y(i)
        enddo
    end subroutine saxpy
end module mymodule

program main
    use cudafor; use mymodule
    real, device :: x_d(2**20), y_d(2**20)
    x_d = 1.0, y_d = 2.0
    ! Perform SAXPY on 1M elements
    call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)
end program main


Standard Fortran

module mymodule contains
    subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        do i=1,n
            y(i) = a*x(i)+y(i)
        enddo
    end subroutine saxpy
end module mymodule

program main
    use mymodule
    real :: x(2**20), y(2**20)
    x = 1.0, y = 2.0
    ! Perform SAXPY on 1M elements
    call saxpy(2**20, 2.0, x, y)
end program main

Parallel Fortran

module mymodule contains
    subroutine saxpy(n, a, x, y)
        attributes(global) subroutine saxpy(n, a, x, y)
            real :: x(:), y(:), a
            integer :: n, i
            attributes(value) :: a, n
            i = threadIdx%x+(blockIdx%x-1)*blockDim%x
            if (i<=n) y(i) = a*x(i)+y(i)
        end subroutine saxpy
end module mymodule

program main
    use mymodule
    real :: x(2**20), y(2**20)
    x = 1.0, y = 2.0
    ! Perform SAXPY on 1M elements
    call saxpy(2**20, 2.0, x, y)
end program main
import numpy as np

def saxpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

cpu_result = saxpy(2.0, x, y)

from copperhead import *
import numpy as np

@cu
def saxpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

with places.gpu0:
    gpu_result = saxpy(2.0, x, y)

with places.openmp:
    cpu_result = saxpy(2.0, x, y)
Dynamic Parallelism

CPU  Non Dynamic Parallelism

CPU  With Dynamic Parallelism
<table>
<thead>
<tr>
<th>Grid Type</th>
<th>Performance</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse grid</td>
<td>Higher Performance</td>
<td>Lower Accuracy</td>
</tr>
<tr>
<td>Fine grid</td>
<td>Lower Performance</td>
<td>Higher Accuracy</td>
</tr>
<tr>
<td>Dynamic grid</td>
<td></td>
<td>Target performance where accuracy is required</td>
</tr>
</tbody>
</table>
What is Dynamic Parallelism?

The ability to launch new kernels from the GPU

- Dynamically - based on run-time data
- Simultaneously - from multiple threads at once
- Independently - each thread can launch a different grid

Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaMemcpySynchronize();
    return 0;
}

__global__ void B(float *data) {
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaMemcpySynchronize();
    do_more_stuff(data);
}
Compiling a CUDA™ code

Using nvcc™ compilator

Visit this site and run the examples (after this session):

https://docs.nvidia.com/cuda/cuda-samples/index.html

Typical compiling

nvcc mycudacode.cu

Specific compilation

nvcc -(args) mycudacude.cu - (extensions)
NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications
Example 1: Hello World

```c
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Instructions:
1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run
CUDA Example 1: Hello World

__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}

Instructions:
1. Add kernel and kernel launch to main.cu
2. Try to build
CUDA Example 1: Build Considerations

- **Build failed**
  - Nvcc only parses .cu files for CUDA

- **Fixes:**
  - Rename main.cc to main.cu
    OR
  - nvcc -x cu
    - Treat all input files as .cu files

Instructions:
1. Rename main.cc to main.cu
2. Rebuild and Run
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

```bash
$ nvcc main.cu
$ ./a.out
Hello World!
```

- `mykernel` *(does nothing, somewhat anticlimactic!)*
The Real and Complete « Hello World » in CUDA

```c
// This is the REAL "hello world" for CUDA!
// It takes the string "Hello ", prints it, then passes it to CUDA with an array
// of offsets. Then the offsets are added in parallel to produce the string "World!"
// By Ingenar Fraenkel 2010

#include <stdio.h>

const int N = 7;
const int blockSize = 7;

__global__
void hello(char *a, int *b) {
    a[threadIdx.x] += b[threadIdx.x];
}

int main() {
    char a[N] = "Hello ";
    int b[N] = {15, 10, 5, 0, -11, 1, 0};

    char *ad;
    int *bd;
    const int csize = N*sizeof(char);
    const int isize = N*sizeof(int);

    printf("%s", a);
    
cudaMalloc((void**)&ad, csize);
    cudaMalloc((void**)&bd, isize);
    cudaMemcpy(ad, a, csize, cudaMemcpyHostToDevice);
    cudaMemcpy(bd, b, isize, cudaMemcpyHostToDevice);

    dim3 dimBlock( blockSize, 1 );
    dim3 dimGrid( 1, 1 );
    hello<<<dimGrid, dimBlock>>>(ad, bd);
    cudaMemcpy(a, ad, csize, cudaMemcpyDeviceToHost);
    cudaFree( ad );

    printf("%s\n", a);
    return EXIT_SUCCESS;
```
Compiler Flags

- Remember there are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code

- NVCC supports some host compiler flags
  - If flag is unsupported, use -Xcompiler to forward to host
    - e.g. -Xcompiler -fopenmp

- Debugging Flags
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information with symbols
CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary
    %> cuda-memcheck ./exe
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck
• nvidia-smi : The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.

• Explore the site: http://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries and follow the instructions for the commands and see the information in the selected node of the practice.
nvidia-smi : The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.

Explore the site: http://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries and follow the instructions for the commands and see the information in the selected node of the practice.
Working at home!

Follow the next tutorial using HPC facilities in the university:

Thank you! @carlosjaimebh