Assignment II: GPU Architecture

Link to GitHub repository: https://github.com/ChrisWe99/DD2360HT22

Exercise 1 – Reflection on GPU-accelerated Computing

1. **List the main differences between GPUs and CPUs in terms of architecture**

GPUs	CPUs
Throughput-oriented architecture relying on 4 architectural features to exploit parallelism (many simple processing units, hardware threads, SIMD execution, simpler memory hierarchy)	Latency-oriented architecture: Minimizing the latency of a single task
Lots of cores, but little hardware for control	Lots of hardware for control; fewer ALUs
Still requires OS, IO and scheduling	Provides OS, IO and scheduling
Lower clock speed, smaller caches (SIMD)	Higher clock speed, larger cache
Excels at regular math-intensive work with little synchronization	Excels at irregular control-intensive work

2. **Check the latest Top500 list that ranks the top 500 most powerful supercomputers in the world. In the top 10, how many supercomputers use GPUs? Report the name of the supercomputers and their GPU vendor (Nvidia, AMD, ...) and model.**

In the Top 500 list of November 2022, 7 supercomputers use a GPU. The only vendors are AMD and NVIDIA.

Rank	Name	GPU Vendor (and model)
1	Frontier	AMD (Instinct MI250X)
2	Supercomputer Fugaku	/
3	LUMI	AMD (Instinct MI250X)
4	Leonardo	NVIDIA (A100 SXM4 64GB)
5	Summit	NVIDIA (Volta GV100)
6	Sierra	NVIDIA (Volta GV100)

7	Sunway TaihuLight	/
8	Perlmutter	NVIDIA (A100 SXM4 40GB)
9	Selene	NVIDIA (A100)
10	Tianhe-2A	/

3. **One main advantage of GPU is its power efficiency, which can be quantified by Performance/Power, e.g., throughput as in FLOPS per watt power consumption. Calculate the power efficiency for the top 10 supercomputers. (Hint: use the table in the first lecture**

The following ranking has been used to calculate the power efficiency:

https://www.top500.org/lists/top500/2022/11/

Formula: Rmax(Pflop/s) / Power (kW)

Rank	Name	Power efficiency (GFlops/Watt)
1	Frontier	52.227
2	Supercomputer Fugaku	14.783
3	LUMI	51.380
4	Leonardo	31.141
5	Summit	14.719
6	Sierra	12.724
7	Sunway TaihuLight	6.051
8	Perlmutter	27.374
9	Selene	23.983
10	Tianhe-2A	3.324

Exercise 2 – Device Query

1. **The screenshot of the output from you running deviceQuery test**

```
!./deviceQuery/deviceQuery_executable

./deviceQuery/deviceQuery_executable Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          11.2 / 11.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15110 MBytes (15843721216 bytes)
  (40) Multiprocessors, ( 64) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 4
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.2, CUDA Runtime Version = 11.2, NumDevs = 1
Result = PASS
```

2. **What architectural specifications do you find interesting and critical for performance? Please provide a brief description.**

- CUDA Driver Version / Runtime Version: This doesn't influence performance from a hardware point of view but newer versions typically implement functions

more efficiently or have more capabilities and can therefore lead to better performance.

- Total amount of global memory: A higher memory allows to store more data at once and improves performance because larger data batches can be handled and accessed directly from the GPU.
- Number of multiprocessors and CUDA cores/MP: Typically, a GPU has multiple multiprocessors (in my case 40 SMs) and each multiprocessor has multiple CUDA cores (in my case 64). The higher the number of cores, the higher the performance since more data can be processed in parallel. More cores increase the theoretical peak throughput (see Exercise 3, question 2)
- GPU Max Clock rate: The clock rate influence how many calculations per second can be achieved. A higher clock rate yields a higher performance (and throughput).
- Memory clock rate: A higher memory clock rate yields a higher memory bandwidth and thus better performance
- Memory bus width: A larger bus width allows more data transfer and consequently better performance. Here, 256 bit per clock memory clock cycle can be transferred.
- L2 cache size: A larger l2 cache size speeds up the memory access since it has a faster access time than e. g. the DRAM. A larger cache leads to better performance.
- Warp size and number of threads: More threads allow a better latency hiding. In general, more threads can allow better latency hiding but this highly depends on the implementation and if more threads are really necessary.

3. **How do you calculate the GPU memory bandwidth (in GB/s) using the output from deviceQuery? (Hint: memory bandwidth is typically determined by clock rate and bus width, and check what double data rate (DDR) may impact the bandwidth)**

Memory bandwidth (GB/s) = (Memory Clock Rate) * (Memory Bus Width) * (DDR factor) = 5001MHz * 256bit * 2 = 2560512bit/s = 320.064 GB/s

The factor 2 is used because of the double data rate (see definition by Intel).

4. **Compare your calculated GPU memory bandwidth with Nvidia published specification on that architecture. Are they consistent?**

Nvidia reports a peak memory bandwidth up to 320 GBytes/s in their datasheet. Consequently, the calculations are consistent with the published specification.

Exercise 3 – Compare GPU architecture

1. **List 3 main changes in architecture (e.g., L2 cache size, cores, memory, notable new hardware features, etc.)**

Some of the newest models from this architecture overview have been chosen.

GPU model	Architecture	GPU memory	GPU Memory Bandwidth	Interconnect bandwidth	L2 cache size
A100 80GB SXM	Ampere	80GB HBM2e	2039 GB/s	NVLink: 600 GB/s; PCIe Gen4: 64 GB/s	40960 KB
Tesla V100 SXM2	Volta	32GB / 16GB HBM2	900 GB/s	300GB/s	6144 KB
H100 SXM	Hopper	80 GB	3352 GB/s	NVLink: 600GB/s; PCIe Gen5: 128 GB/s	~50MB

Notable hardware features:

- Ampere: Third generation NVLink that doubles the GPU-to-GPU direct bandwidth to 600GB/s
- Volta: Includes optimized software for mixed precision
- Hopper: Transformer engine for Tensor Cores to speed up AI model training

2. **List the number of SMs, the number of cores per SM, the clock frequency and calculate their theoretical peak throughput.**

Formula: (#SMs) * (#cores per SM) * (clock frequency) * (FLOPS per cycle)
Flops per cycle are assumed to be one in the calculations. (Note: The previously linked datasheets also allow to check for the provided peak TFLOPS which are often a multiple of the calculated theoretical peak throughput below due to the number of calculations per cycle which depends on the type e. g. FP64, FP32, …).

Model	#SMs	#Cores per SM	Clock frequency	Theoretical peak throughput (TFLOPS)
A100 80GB SXM	108	FP64: 32 FP32: 64 INT32: 64 Tensor Cores: 4	1410MHz	4.873 9.746 9.746 0.610
Tesla V100 SXM2	80	FP64: 32 FP32: 64 INT32: 64 Tensor Cores: 8	1530MHz	3.917 7.834 7.834 0.979
H100 SXM	132	FP64: 32 FP32: 128 INT32: 64 Tensor Cores: 4	1980 MHz	8.364 33.454 16.727 1.045

Sources:

- Ampere: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
- Volta: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
- Hopper: https://resources.nvidia.com/en-us-tensor-core

3. **Compare (1) and (2) with the NVIDIA GPU that you are using for the course.**

For this course, I'm using Google Colab and the assigned GPU on the platform is a NVIDIA Tesla T4 (based on Turing architecture).

GPU model	Architecture	GPU memory	Interconnect bandwidth	L2 cache size
Tesla T4	Turing	16GB GDDR6	32 GB/s	~4 MB

The used GPU is obviously less powerful than the presented ones before.

Model	#SMs	#Cores per SM	Clock frequency	Theoretical peak throughput (TFLOPS)
Tesla T4	40	CUDA Cores: 64 Tensor Cores: 8	1590MHz	4.070 0.509

Also here, the T4 has different cores per SM and the theoretical throughput is lower than for most of the other GPUs.

Source for Tesla T4: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Exercise 4 – Rodinia CUDA benchmarks and Profiling

1. **Compile both OMP and CUDA versions of your selected benchmarks. Do you need to make any changes in Makefile?**

I tried out every single benchmark and decided to use a benchmark which does not require an input file (but only arguments) and which has the timing already built in.

The main error every time was the used GPU architecture. Since the Google Colab GPU is a Turing Architecture, this website helped me to get the correct configuration.

Benchmark 1: Lud

For the CUDA version, I received a GPU architecture error and had to change 'sm_13' to a valid architecture for the NVIDIA Tesla T4 which is 'sm_75'

For the OMP version, I got the error 'icc: Command not found'. To solve this error and the following ones, I changed the compiler in the Makefile.offload.
CC:= icc was changed to CC:=gcc
CXX:= icc was changed to CXX:=gcc

Benchmark 2: Myocyte

I didn't have to change anything here.

Benchmark 3: LavaMD:

I received the GPU architecture error:
Value 'sm_13' is not defined for option 'gpu-architecture'.
Consequently, I had to change the value in the Makefile to a suitable value for the Tesla T4 which is 'sm_75'.

2. **Ensure the same input problem is used for OMP and CUDA versions. Report and compare their execution time.**

Benchmark	Input problem	CUDA time	OMP time
Lud	!./cuda/lud_cuda -s 2500 !./omp/lud_omp -s 2500	94.997 ms	1300.342 ms
Myocyte	!./myocyte.out 2000 200 1 !./myocyte.out 2000 200 1 0	16.434 s	59.183 s
LavaMD	!./lavaMD -boxes1d 15 !./lavaMD -cores 8 -boxes1d 15	0.0693 s	22.584 s

For all 3 selected benchmarks, larger input problems lead to a larger gap between CUDA and OMP execution time.

3. **Do you observe expected speedup on GPU compared to CPU? Why or Why not?**

Yes, I observe a clear speedup for all the benchmarks. The reason for that is that the input problems are large enough to visibly benefit from the possibility of parallelization for the different problems.

For Lud, the CUDA version takes only 7.3% of the execution time of OMP.

For Myocyte, the CUDA version requires only 27.77% of the execution time of the OMP version.

For LavaMD, the CUDA version is extremely faster. It needs 0.3% of the execution time of the OMP version although already the maximum number of cores is configured for the OMP execution.

(Bonus) Exercise 5 – GPU Architecture Limitations and New Development

Chosen publication: *Choukse, E., Sullivan, M. B., O'Connor, M., Erez, M., Pool, J., Nellans, D., & Keckler, S. W. (2020, May). Buddy compression: Enabling larger memory for deep learning and HPC workloads on GPUs. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) (pp. 926-939). IEEE.*

1. **What limitations this paper proposes to address**

Many high-throughput applications like typical applications for HPC or DL models require large memory space and high memory bandwidth on the GPU. However, memory with high bandwidth is typically relatively small. And memory that would be large enough, typically has lower bandwidth.

Hence, currently, developers have to use workarounds like processing multiple smaller batches in a serial manner which can result in low utilization and accuracy issues. Or they have to choose from other alternatives like data movement orchestration between CPU and GPU, the usage of multiple GPUs or the usage of Unified Memory to handle larger data packages. All these alternatives have major drawbacks which makes them unattractive options to compensate limited GPU memory.
Additionally, memory compression on GPUs usually leads to problems: If data compressibility changes over time, re-allocation of memory or page movement is required on GPUs. Thus, mainly domain specific compression but not general-purpose compression has been studied.
The presented concept, Buddy Compression, solves both problems (larger memory with sufficient bandwidth and no re-allocation for different compression sizes) and is explained in more detail in question 3.

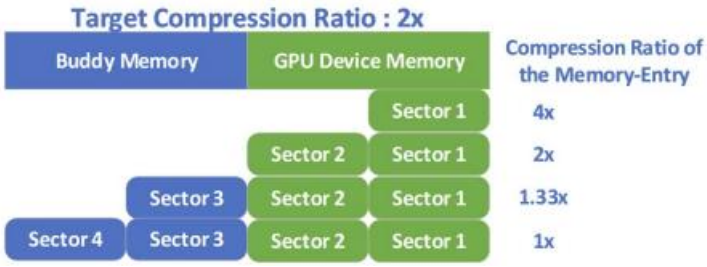
2. **What workloads/applications does it target?**

It targets applications that require larger GPU memory. This can be HPC or DL workloads that benefit from an expanded GPU memory capacity.

3. **What new architectural changes does it propose? Why it can address the targeted limitation?**

This work proposes Buddy Compression in order to address the mentioned limitations. Buddy Compression is a general-purpose mechanism that increases user-visible memory capacity on GPUs. In general, it leverages high-bandwidth interconnects and data compressibility.
The Buddy Compression framework allows the developer to execute memory allocations that use less device memory than the actual allocation size. For example, if 12GB of data should be allocated on a GPU with 6GB memory capacity, the data can be allocated with a target of 2x compression. Consequently, only half of the 12GB is allocated on the GPU memory. However, if the data can’t be compressed sufficiently to 6GB, a high-bandwidth interconnection (e.g. NVLink2) to a larger but slower buddy-memory is used as a overflow storage and enlarges the from the GPU accessible memory. E.g. if the compression rate is 1.5 instead of the targeted 2x, the

resulting 8GB would be split up: 6GB to the GPU memory and the remaining 2GB are placed in the Buddy Memory.
As a compression algorithm to achieve the required compression rates, several low-cost methods have been compared and Bit-Plane Compression was evaluated as the most attractive one for Buddy Compression by the authors.
The subsequent figure (from the paper) illustrates the concept again: A 128Byte memory-entry is compressed by the factor that is denoted on the right of the figure. One sector can store 32 Byte. If the target compression ratio of 2 is not achieved, additional sectors on the Buddy Memory are used to store the full data and still have it accessible as fast as possible for the GPU.



According to the authors, the Buddy Compression achieves a 1.5 to 1.9x GPU capacity expansion for the evaluated applications that are described in the next question. Moreover, it only performs 1-2% worse than an unconstrained-capacity GPU (which is not possible in reality).

4. **What applications are evaluated and how did they setup the evaluation environment (e.g., simulators, real hardware, etc)?**

Workloads:
For performance evaluation, they use a set of 10 HPC and 6 DL network training workloads.
In more detail, they choose simulatable representatives for HPC applications from a subset of SpecACCEL OpenACC v1.2 and CUDA versions of the DOE benchmarks HPGMG and LULESH.
For the DL workloads, they train five different convolutional neural networks: AlexNet, Inception v2, SqueezeNetv1.1, VGG16, and ResNet50. All of them are running on the Caffe framework and will be trained on the ImageNet dataset. Moreover, the sixth DL workload is a long short-term memory network, called BigLSTM, that uses the English language model.

Simulation infrastructure:

As a simulation environment, the authors use a dependency-driven GPU performance simulator. Its configuration is based on public information about NVIDIA's P100 Pascal GPU and the interconnect characteristics of recent Volta GPUs. The SMs are modeled as an in-order processor with the scheduling technique greedy-then-oldest warp. Caches are included as multi-level hierarchy with private L1 caches and a shared sectored L2 cache.

In order to evaluate the Buddy Compression performance, the authors compare it to a unconstrained-capacity GPU (in theory) as well as with established existing methods that are used to run the applications (e.g. Unified Memory technique).

5. Do you have any doubts or comments on their proposed solutions?

I don't have doubts that the proposed method can speed up applications by extending the GPU memory with a buddy memory. However, this speedup heavily depends on the interconnection to the buddy memory. If the hardware connection is a slow one (e. g. PCIe or even worse), the required access time would be higher and the speedup less relevant or not even existent.