Assignment 2: Accelerator-Based Programming

Misael Jordan Enrico

12 February 2024

1 Task 1

The matrix-vector product between matrix `mat` and vector `vec_in` has been created to calculate vector `vec_out`. The matrix is stored in row-major format. The kernel launch configuration is as follows:

- Block size : 1024
- Grid size : `(M + WARPS_PER_BLOCK - 1) / WARPS_PER_BLOCK`

In this case, `WARPS_PER_BLOCK` is 32 since the block size is set to 1024. With this configuration , I aim to assign one dot product calculation (a product of one row of the matrix with `vec_in`) to be handled by one warp. All threads within a block will populate the shared memory `shmem_vec_in`using the values from `vec_in`. The way shared memory is populated is that adjacent threads within a block read adjacent values from `vec_in`to ensure coalesced memory access pattern. At the end of this step, the shared memory will be of size `N`. After shared memory is populated, each warp within a block will perform vector-vector dot product between a particular row of the matrix that it is assigned to and vector `vec_in`. Again, adjacent threads within a warp perform multiplication operation where the operands are adjacent from each other. When the number of columns are more than warp size (32), it needs to perform this multiplication operation again until `N` multiplications are performed. These steps are illustrated on figure 1. At the end, we have 32 summation values from each threads.

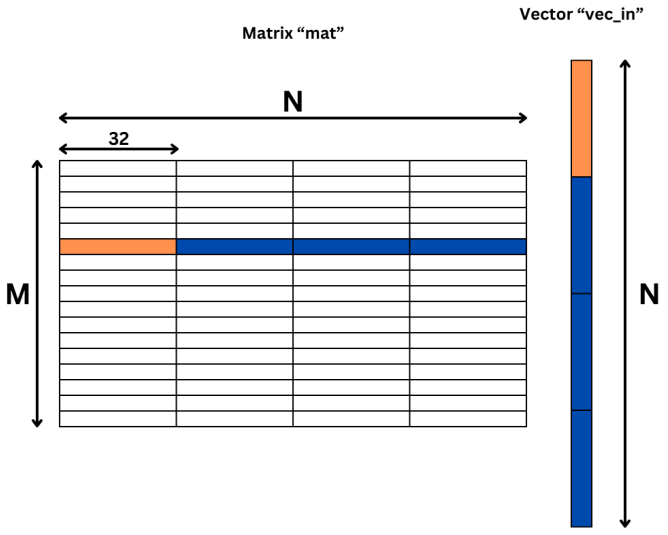

Figure 1: Calculation performed by a warp being highighted in color.

Each thread within a warp has a variable called `tmp` in the register which represents partial sum calculated by that particular thread. We perform sum reduction across all `tmp`s on threads at the warp level using the `__shfl_xor_sync()` function. Finally (only) the first thread in the warp writes its value to the resulting vector `vec_out`. From my observation, `__shfl_xor_sync()` greatly reduces the clock cycle it takes to write back to `vec_out`.

For the naive approach, each thread calculates a single entry for output vector `vec_out`. This is to show how poor the kernel is. Block size is set to 32 because if we go lower than that, warp scheduler can't fully schedule a full warp, which is of size 32 threads. Shared memory is not utilized to store the input vector `vec_in`. I think the read access of threads within a block still coalesced however, read access to the matrix is definitely not. Adjacent threads are reading data from the matrix `mat` that are separated `N` floats apart for each iteration.

1.a Bandwidth Measurement Between `N=100` until `N=10000`

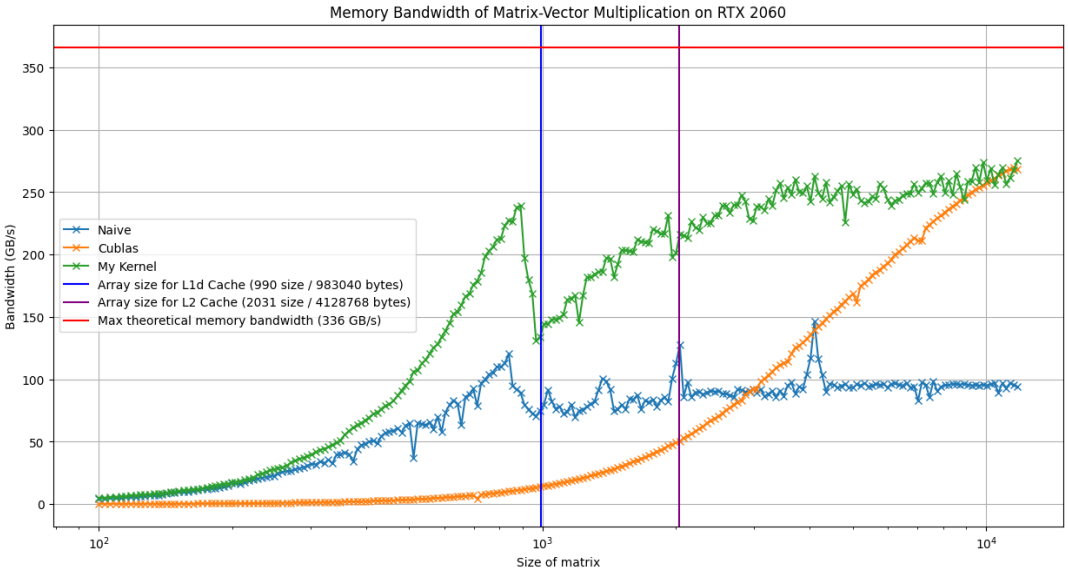

Figure 2: Memory bandwidth of the Matrix-Vector multiplication on Nvidia RTX 2060.

1.b Kernel Configuration

For large matrix such as when `N=8000`, my kernel needs more than half of the shared memory (65536 bytes). This means, only one block can execute at a time in an SM. Therefore, I decided to fully use all 1024 threads in a single block to maximize occupancy (maximum number of threads in Nvidia RTX 2060 is 1024). Lowering block size to 512 threads is not beneficial when heavy usage of shared memory prevents us from issuing 2 blocks in an SM. The other 512 threads will be idle (50% occupancy). The downside is that for smaller sizes when the number of rows `M`, not all SMs are utilized. There are 30 SMs in Nvidia RTX 2060. This means the number of rows has to be at least $30 \times 32 = 960$ for the GPU to use all of its SMs. The solution would be to assign more warps to calculate a single entry of the output vector rather than just 1. In the case of Nvidia RTX 2060 with 30 SMs, if we assign 2 warps while maintaining the same kernel launch configuration, a block can handle 16 output entries. This way, the number of rows needed to utilize all SMs are halved to $30 \times 16 = 480$. In essence, my kernel will fully utilize all SMs when number of rows is greater than 960 and fail when number of columns is greater than 16384 due to insufficient shared memory size.

For naive kernel, each row is handled by one thread. This means, it's most likely not using the entire SM's available resources when the number of row is less than $30 * 1024 = 30720$. We expect a very low bandwidth for this kernel.

2 Task 2

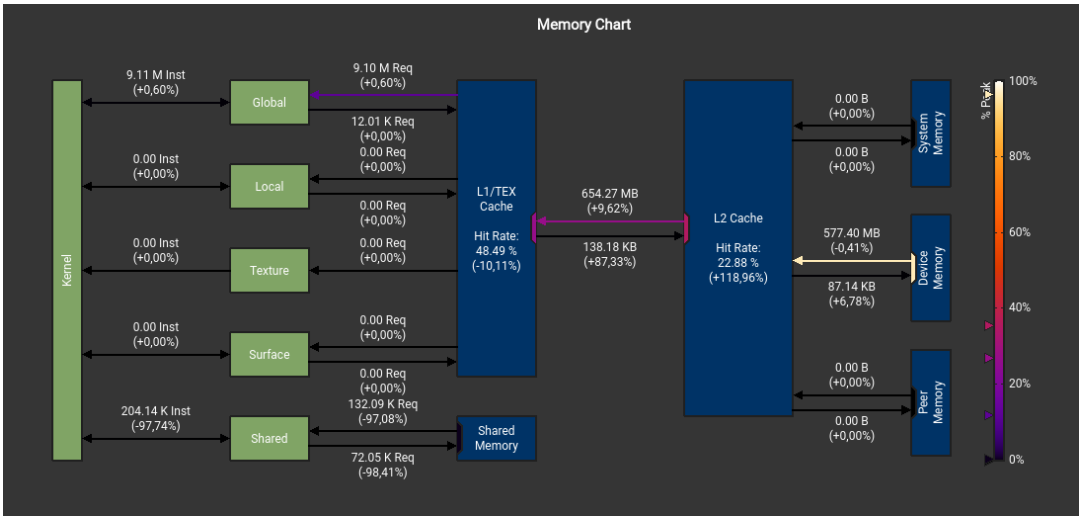

Figure 3: Memory chart of `cublasSgemv()` kernel with my kernel as baseline. Notice that the load/store requests to/from shared memory is 97-98% lower than my kernel. Diagram is obtained from Nsight Compute profiler.

2.a Performance Differences Compared to CUDA

From figure 2, my kernel seem to achieve higher bandwidth throughout the test compared to the cublas version. For large `N`, the bandwidth is comparable to cublas. However, my implementation fails once `N` reaches 16384 because of full shared memory. Cublas implementation doesn't utilize shared memory intensively as seen on figure 3. The `cublasSgemv()` kernel sends around 97% less load/store shared memory instructions. To test whether `cublasSgemv()` is indeed IO bound, we need to run it for larger input size.

2.b Number of Columns is Kept Constant at 10000

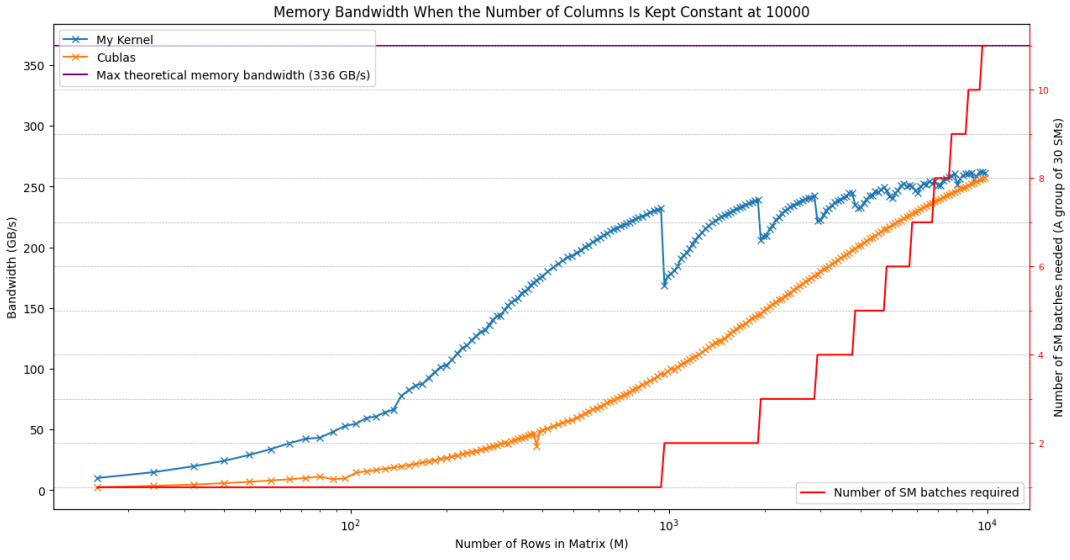

Figure 4: Memory bandwitdh of the kernel and cublas `cublasSgemv()` when number of columns is constant at 10000.

There is a shark-fin like pattern on the bandwidth graph of my kernel. When the number of threads (or blocks) issued is larger than all 30 SMs can run at a given time, the remaining threads must wait until there is an SM that

can handle it. In my kernel, the block size is 1024 and each block calculates 32 rows of the matrix. There are 30 SMs in the GPU with each only capable of handling one block at a time. If the matrix has more than 960 rows, there will be one block that has to wait for first 32 blocks to finish before it starts running. I think this is why my kernel first drop in bandwidth at 960 rows mark. The number of blocks batch required is drawn in red in diagram 4.

Different from my implementation, I think `cublasSgemv` has a smaller block size launch configuration. It may be experiencing the same condition as what my kernel experienced but hidden due to smaller blocks granularity.

2.c Number of Rows is Kept Constant at 16384

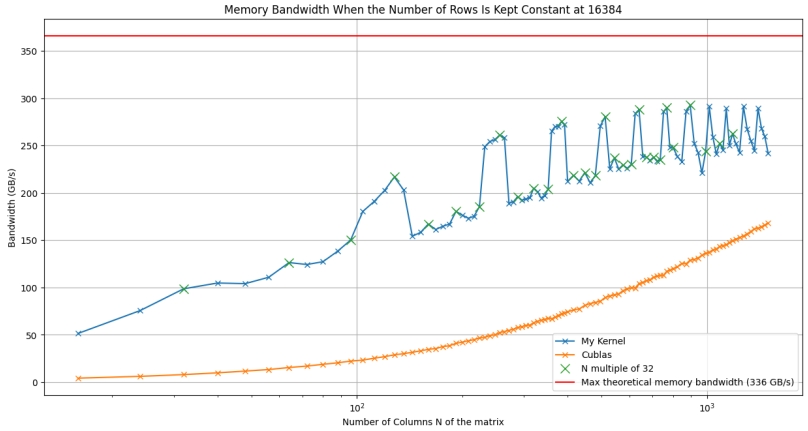

Figure 5: Memory bandwitdh of the kernel and cublas `cublasSgemv()` when number of columns is constant at 16384.

Again, my kernel seem to be faster within the range of `N<1500`, but is limited to $N = 16384$ due to shared memory limitation. I observed a strange phenomena where my kernel's bandwidth fluctuates heavily while increasing. At first, I thought this is due to the number of columns not being a multiple of warp size (marked in green X on figure 5. However, upon investigation, this is proved to be not the case as bandwidth can still fluctuate regardsless of whether it's a multiple of 32 or not. The `nvidia-smi` command tells us that there are no other processes running in the back ground. For this reason, I think this is the property of my kernel that I still don't know what the reason is.

3 Task 3

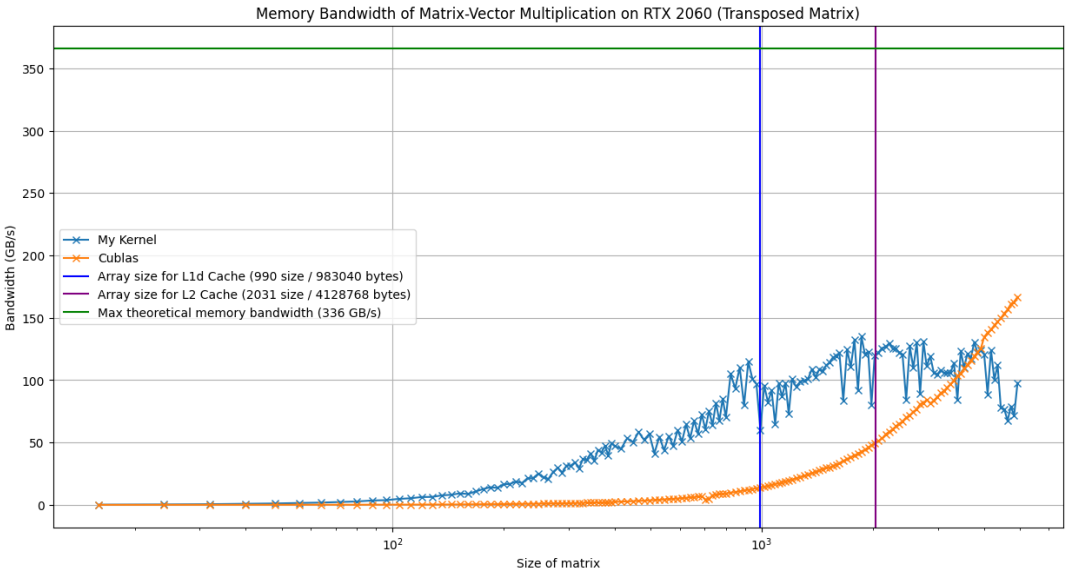

Figure 6: Memory bandwitdh of the kernel and cublas `cublasSgemv()` when matrix is transposed.

My implementation of matrix-vector multiplication with transposed matrix `mat` works the same as the previous section. The difference is that now each warp is assigned to a column of that matrix. The reason for this is that now the matrix can be thought as in column-major format. As expected, this kernel performs worse than the non-transposed version because there is a major uncoalesced memory access pattern. Adjacent threads within a warp now always access different values in memory that's separated `N` floats away. As seen on figure 6, my kernel seem to be faster for smaller matrix size. For small matrix size, this uncoalesced pattern may not be identifiable from the bandwidth graph alone. As size increases, the uncoalesced access issue becomes a problem and slows the kernel run time. The cublas version however manage to achieve the same peak matrix-vector multiplication bandwidth just like the non-transposed version. My kernel needs to be improved in terms of memory access pattern. A simple solution would be to transpose the matrix before performing the multiplication.

In my implementation, there are two locations where race condition might occur. The first one is when `__shfl_xor_sync()` is called to obtain reduced sum value within a warp. Function `__shfl_xor_sync()` has implicit barrier `__syncwarp()` which tells all threads within warp to not continue before all threads are done. The second race condition location is when the vector-vector multiplication value is written back to the output vector `vec_out`. This is not an issue since one element of the output vector is handled by one warp only. Race condition may occur at this point if we assign two warps or more to calculate one entry of output vector. Multiple warp could possibly try to update the value of output vector at the same time.

4 Task 4

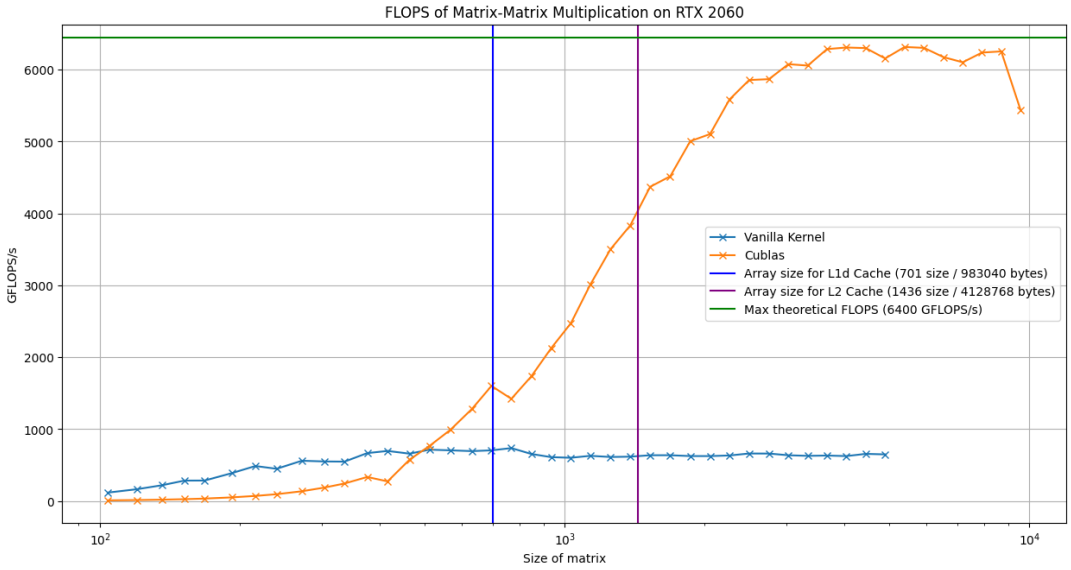

Figure 7: Memory bandwitdh of the kernel and cublas `cublasSgemv()` when matrix is transposed.

Here, matrix multiplication between `A` and `B` is performed using a naive kernel and cublas's `cublasSgemm()` function. For the naive kernel, each thread calculates an element of the output matrix without utilizing shared memory. Assuming threads are ordered in a row-major order, adjacent threads read adjacent values from matrix `A` (broadcast access). Therefore memory access is coalesced. Memory access pattern to matrix `B` is coalesced as well for the same reason (non-broadcast access). However, the performance of the naive kernel is extremely poor compared to `cublasSgemm()`. This is because shared memory is not utilized. Adjacent threads use the same row from matrix `A` to perform its calculation and therefore should be put into the shared memory. This lowers the number of float read by $(N-1) \times N$ where N is the number of columns of matrix `A`. It leveled off at around 620 GFLOPS/second while cublas performs 10 times better at around 6000 GFLOPS/second and peaked at 6316.38 GFLOPS/second when $N = 5392$. For the cublas version, there's a slight decrease when the matrix size hits the L1 cache limit. This might indicate that for smaller matrix size, cublas is still memory bound. Its peak FLOPS is close to the maximum theoretical GPU FLOPS when $N >= 3672$. For this reason, I can conclude that the cublas version is compute bound when the input size is large.

5 Hardware Specifications

5.a Remote Server

All programs specifically instructed to be run on CPU are run on Snowy (`snowy.uppmax.uu.se`) and Vitsippa servers (`vitsippa.it.uu.se`). Snowy uses Intel Xeon E5-2630 while Vitsippa runs on AMD Opteron 6282 SE.

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	48 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	40
On-line CPU(s) list	0-39
Vendor ID	GenuineIntel
Model name	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
CPU family	6
Model	79
Thread(s) per core	2
Core(s) per socket	10
Socket(s)	2
Stepping	1
BogoMIPS	4390.24
Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid amd_dcm aperfmperf pni pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 popcnt aes xsave avx lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt fma4 nodeid_msr topoext perfctr_core perfctr_nb cpb hw_pstate ssbd ibpb vmmcall arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	25600K

Table 1: The specification of the CPU of Snowy server.

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	48 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	32
On-line CPU(s) list	0-31
Vendor ID	AuthenticAMD
Model name	AMD Opteron(tm) Processor 6282 SE
CPU family	21
Model	1
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	2
Stepping	2
BogoMIPS	5200.21
Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid amd_dcm aperfmperf pni pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 popcnt aes xsave avx lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt fma4 nodeid_msr topoext perfctr_core perfctr_nb cpb hw_pstate ssbd ibpb vmmcall arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold
Virtualization	AMD-V
L1d cache	512KiB (32 instances)
L1i cache	1MiB (16 instances)
L2 cache	32MiB (16 instances)
L3 cache	24MiB (4 instances)

Table 2: The specification of the CPU of Vitsippa server.

5.b Local Machine

All programs which use GPU are run on a local machine which runs on `TTY1` to minimize background processes. It uses AMD Phenom II X4 965 Processor and NVIDIA GeForce RTX 2060 GPU.

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	48 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	4
On-line CPU(s) list	0-3
Vendor ID	AuthenticAMD
Model name	AMD Phenom(tm) II X4 965 Processor
CPU family	16
Model	4
Thread(s) per core	1
Core(s) per socket	4
Socket(s)	1
Stepping	3
BogoMIPS	6800.21
Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext 3dnow constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt nodeid_msr hw_pstate vmmcall npt lbrv svm_lock nrip_save
Virtualization	AMD-V
L1d cache	256 KiB (4 instances)
L1i cache	256 KiB (4 instances)
L2 cache	2 MiB (4 instances)
L3 cache	24 MiB (4 instances)

Table 3: The specification of the CPU of the local server.

Device	NVIDIA GeForce RTX 2060
CUDA Driver Version / Runtime Version	12.3 / 12.3
CUDA Capability Major/Minor version number	7.5
Total amount of global memory	5926 MBytes (6213402624 bytes)
(030) Multiprocessors, (064) Cuda Cores/MP	1920 CUDA Cores
GPU Max Clock rate	1680 MHz (1.68 GHz)
Memory Clock rate	7001 Mhz
Memory Bus Width	192-bit
L2 Cache Size	3145728 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	Disabled
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 / 1 / 0

Table 4: The specification of the GPU of the local server.