Performance Comparison of Different GPU Hardware Settings

Nicolas Kozachuk

Department of Electrical & Computer Engineering

Lehigh University

ngk324@lehigh.edu

Date: May 10th, 2024

Abstract

This report provides a comprehensive analysis of the performance comparison of GPU speedup on three different hardware settings. The three different hardware settings used include a single GPU single stream, single GPU concurrent streams, and multiple GPUs. Each of these different hardware settings were applied to three different codes in order to produce a comprehensive performance comparison. This was done by modifying previous code in order to turn the code into CUDA C/C++ in order to apply the different hardware settings. The different codes used include a code that performs doubling a vector, code that performs 2D matrix multiplications, and a code that performs vector addition. The code was then run on the Lehigh University ECE High-Performance Computing(HPC) clustering and analyzed using NVIDIA Profiler tools. All in all, this report provides a comprehensive performance comparison of GPU speed amongst the hardware settings of single GPU with single stream, single GPU with concurrent streams, and multiple GPUs.

Introduction, Background, and Theory

The purpose of this report is to provide a performance comparison of different GPU hardware settings. The GPU hardware settings analyzed in this report include single GPU single stream, single GPU concurrent streams, and multiple GPUs. Each of these settings were implemented for three different types of code, which include code that performs doubling of vector elements, code that performs 2D matrix

multiplication, and a code that performs vector addition. The different GPU hardware settings were implemented with the code using CUDA C/C++ and ran on the Lehigh HPC.

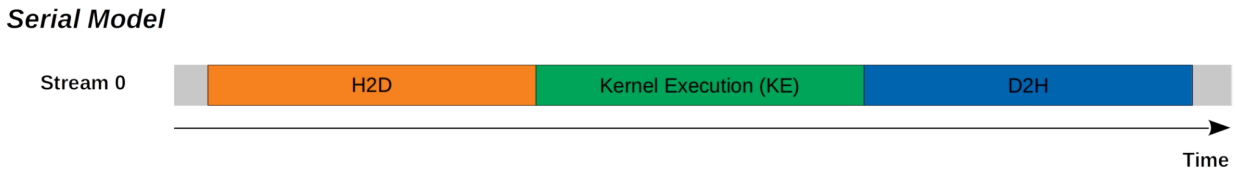


Figure 1: Single GPU with Single Stream [1]

A single GPU with a single stream is the default hardware setting when utilizing GPUs. How a single GPU with a single stream works can be seen above in Figure 1. It can be seen that the GPU works serially, performing one task at a time with no overlapping. The example in Figure 1 shows that first in the single GPU single stream, a host(CPU) to device(GPU) memory transfer is performed, as can be seen by the orange H2D block. Next the GPU performs a kernel execution, as can be seen in the green block. The kernel execution is the computational task that is performed by the GPU. Lastly, a device to host memory transfer is performed, as can be seen by the blue D2H block. This is the standard model when utilizing a single GPU with a single stream.

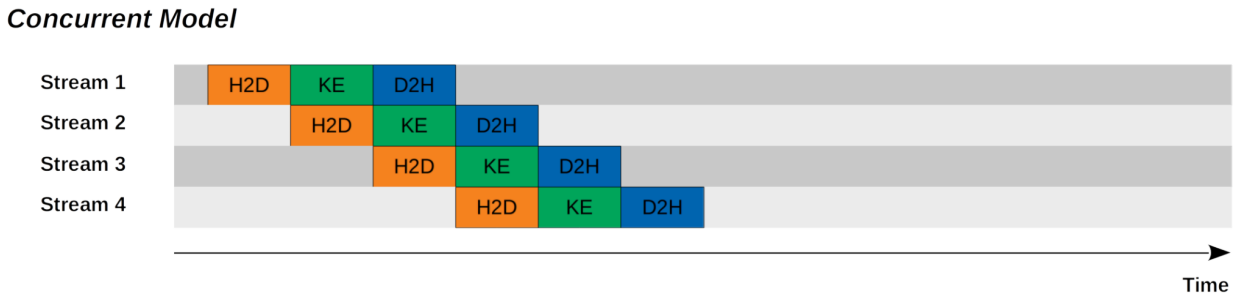


Figure 2: Single GPU with Concurrent Streams [1]

The number of streams used is a GPU hardware setting that can be altered. An example of utilizing a single GPU with four concurrent streams can be seen above in Figure 2. Concurrent streams on a GPU

allow multiple kernels, which perform computational tasks, to execute simultaneously on the device. This capability is particularly valuable for exploiting the massive parallelism inherent in GPU architectures. As can be seen in Figure 4, tasks not utilizing the same resources, such as device and host memory, the ALU, etc., can be performed concurrently by the GPU. For example, in the first step, stream 1 performs a hardware to device transfer. Only stream 1 is able to perform a hardware to device transfer in the first step because host memory can only be accessed by a single stream at a time. The concurrency can be seen in the next step, where stream 1 then performs a kernel execution concurrently with stream 2 performing a host to device memory transfer. Concurrent streams can be seen working best in step 3, where stream 1 performs a device to host memory transfer, stream 2 performs a kernel execution, and stream 3 performs a host to device memory transfer. As can be seen in Figure 2, this trend continues until all streams are done performing their respective tasks, in a concurrent manner. Comparing a single GPU with concurrent streams to a single GPU with single stream, it is clear just from the model of the hardware setting that having concurrent streams will have better performance than a single stream, if not the same for some specific instances, since tasks involved when using a GPU can be performed in parallel when using concurrent streams, but not with a single stream.

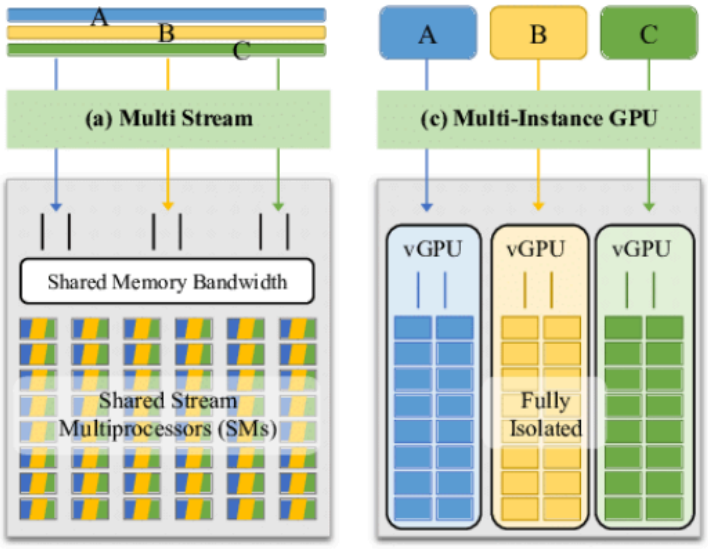


Figure 3: GPU with Concurrent Streams and Multiple GPU Diagram [2]

The last hardware configuration implemented for performance comparison is utilizing multiple GPUs. As can be seen in Figure 3, this is different from concurrent streams, as concurrent streams exist within a singular GPU, while using multiple GPUs uses multiple different isolated GPUs. It is possible though for there to be concurrent streams within multiple GPUs, but this hardware configuration was not implemented in this project. Since concurrent streams exist within a singular GPU, concurrent streams have a shared memory, while multiple GPUs have their own individual memory as each GPU is fully isolated from each other. This is one thing that must be taken into consideration when choosing a hardware configuration and when using multiple GPUs, as when using multiple GPUs, one must perform memory management between the multiple GPUs in order to take full advantage of the performance gains when utilizing multiple GPUs. All together, utilizing a hardware configuration of single GPU with concurrent stream is most advantageous for single GPU setups and when there is significant data movement, while multiple GPUs are most advantageous when there is high computational demand, large-scale workloads, and scalability in memory and compute resources.

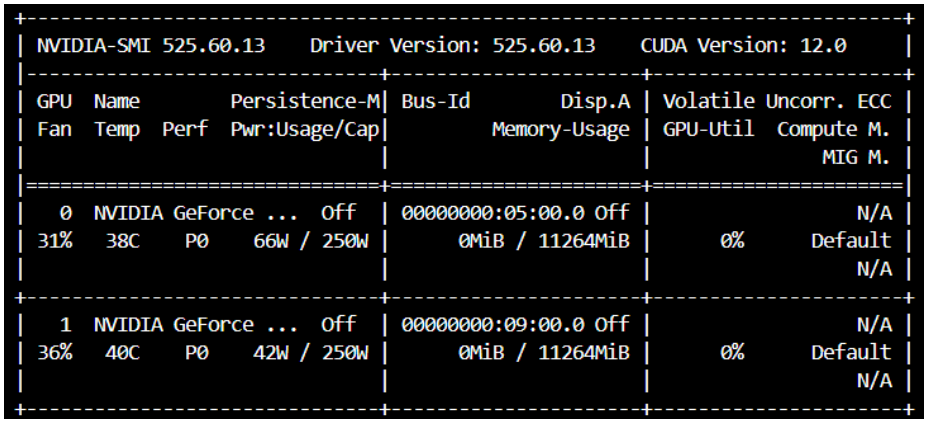


Figure 4: Lehigh HPC GPU Information

Three different types of codes were implemented for each of the different hardware configurations in order to provide a comprehensive performance comparison. The first code is a program that doubles the

elements of a vector. The second code is a program that performs vector addition. The last code is a program that performs 2D matrix multiplication using vectors. These three codes were chosen to cover the range of the arithmetic operations that can be performed on a vector, as well as the dimension of a 2D vector, which contains some data dependencies. These codes were run on the Lehigh HPC in order to utilize multiple powerful GPUs. The Lehigh HPC GPU information can be seen above in Figure 4. The Lehigh HPC has 2 NVIDIA GeForce 12GB GPUs and CUDA version 12.0 is used. These 2 GPUs are what are utilized in order to provide the performance comparison with multiple GPU hardware settings.

Procedure

The first step of this project is to make sure one has access to the Lehigh HPC and that CUDA toolkit works properly. To do this, first get access to the HPC by contacting the Lehigh system administrator. After obtaining an account, ssh into the HPC using username@ece-hpc0?.cc.lehigh.edu, replacing the username and HPC number in the command. After seeing that one can access the HPC successfully, one has to run the commands "source /localproject/setup-info/NVidia-HPC-SDK/NVidia-environment-scripts/NVidia-All-HPC-SDK-23.1-And-Cuda-SDK-12-And-OpenMPI-23.1-SDK-env.sh" and then "scl enable devtoolset-7 bash" each time one accesses the HPC and wants to use the CUDA toolkit. One can run the command "nvcc --version" to check that the CUDA toolkit is working properly. After ensuring that the HPC CUDA environment is working properly, one can open, edit and run code remotely using VS Code. First open VS Code, install the shh extension if it is not already installed, and select the "connect to host" option from the bottom left corner of VS Code. Next, select "add new ssh host" and type "ssh -XY username@ece-hpc0?.cc.lehigh.edu" replacing the username and HPC number in the command. If one is off campus, make sure the Lehigh VPN is turned on.

After successfully completing the previous portion, one can now begin coding. One can first begin by writing three general simple codes that perform vector addition, doubles a vector value, and perform 2D

multiplication using vectors. This project requires basic knowledge of how to utilize CUDA C/C++ code, as well as current streams in GPUs and how to utilize multiple GPUs. To gain basic knowledge of CUDA C/C++ code to accelerate applications, one can go through the NVIDIA course “Fundamentals of Accelerated Computing with CUDA C/C++” [3]. To gain knowledge of how to utilize GPUs with concurrent streams, one can go through the NVIDIA course “Accelerating CUDA C++ Applications with Concurrent Streams”[4]. To gain knowledge of how to utilize multiple GPUs, one can go through the NVIDIA course “Scaling Workloads Across Multiple GPUs with Cuda C/C++”[5]. After one gains this knowledge, they can begin implementing the program acceleration techniques on the previous code written. To write single GPU single stream code, one must first ensure the function performing the computational task is a kernel by adding the “__global__” tag to the beginning of the function declaration. One can then use cudaMallocManaged(variable, size) to allocate memory for a defined size for a variable. Then one must define the number of threads per block and the number of blocks for the kernel, and then the kernel can be called. One must make sure to have a check for any errors that possibly arose in the kernel execution, as well as a call to the cudaDeviceSynchronize() function to ensure all the GPU threads finished completing and call cudaFree(variable) to free the allocated memory. An example of these important parts of code can be seen in Appendix A 1.1. This approach should be used for all three different codes.

Next one can implement using a single GPU with concurrent streams on the three different codes. Like previously, the kernel should be appropriately defined using __global__. One should begin by allocating memory for gpu and cpu variables, define the number of threads per block and the number of blocks for the kernel, define the number of streams and then create an array of defined streams and get the amount of data per stream(Appendix A 2.1). One should then create a for loop that loops through the streams. The beginning of the for loop should calculate the width of the data for the stream, and then asynchronously copy memory from the host to the device. Then the kernel can be called, with the stream specified based on the loop iteration, and then one must check for any errors that possibly arose in the kernel execution, as

well as call cudaDeviceSynchronize() function to ensure all the GPU threads finished completing. This example code can be seen in Appendix A 2.2. One must also remember to call cudaFree(variable) in order to free allocated memory. This approach should be used on all three codes in order to implement a single GPU with concurrent stream versions of them.

Lastly, one can then implement the codes using multiple GPUs. Like previously, the kernel should be appropriately defined. One should begin the code by using cudaGetDeviceCount() to get the number of GPUs, define the number of threads per block and the number of blocks for the kernel, get the amount of data per GPU, and then initialize a pointer array for GPU data. A for loop should then be used to allocate a chunk of memory for each GPU, as can be seen in the code example in Appendix A 3.1. Next, a for loop should be used to copy CPU data to the GPU data array using the cudaDeviceMemCpy() function(Appendix A 3.2). Next a for loop can be created to perform the kernel execution, but first in the while loop the current GPU in use should be specified using the cudaDeviceSet() function, and the width of the data for the GPU should be calculated. Then the kernel can be called using the width calculated and the data specific to the GPU in order to be executed(Appendix A 3.3). One must again check for any errors that possibly arose in the kernel execution, as well as call cudaDeviceSynchronize() function to ensure all the GPU threads finished completing. Lastly the memory should be copied from the GPU array to the CPU array, which can be done using a for loop that calls cudaDeviceMemCpy(), as can be seen in Appendix A 3.4. One must again remember to call cudaFree(variable) in order to free allocated memory. This approach should be used on all three codes in order to implement multiple GPU versions of the codes.

Once all the codes are written, they can be run using the command “nvcc -o executable_name program_name.cu -run”. Additionally, an NVIDIA Program Profile can be generated to get the execution information of the program using the command “nsys profile --stats=true -o report_name executable_name”. From the code's execution profile information, one can see important information

about the performance of the program, such as kernel execution time, memory transfers, API calls, etc. One should profile each of the codes using each of the different versions of the codes for each hardware setting and record their performance. One should then analyze the performance of each hardware setting amongst each of the code and compare their performances.

Results and Analysis

Each of the three different codes were implemented using each of the three different hardware settings. The resulting codes were then profiled using the NVIDIA Profiler tool in order to analyze the code's execution and perform a performance comparison. The first code that was analyzed for the different hardware settings was the double vector elements code. The program was initialized to have vectors of 100,000 int elements.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	980,771	1	980,771.0	980,771.0	980,771	980,771	0.0	32	1	1	1024	1	1	doubleElements(int *, int)

Figure 5: Profile for Double Vector Elements Code with Single GPU with Single Stream

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	10,177	2	5,088.5	5,088.5	4,704	5,473	543.8	32	1	1	1024	1	1	doubleElements(int *, int)

Figure 6: Profile for Double Vector Elements Code with Single GPU with Concurrent Streams

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	11,104	2	5,552.0	5,552.0	4,992	6,112	792.0	32	1	1	1024	1	1	doubleElements(unsigned long *, int)

Figure 7: Profile for Double Vector Elements Code with Multiple GPUs

One can see the profile results for this code when using a single GPU with a single stream in Figure 5, which shows that the execution time of the double elements kernel took 980,771ns. One can also see the profile results for this code when using a single GPU with multiple streams and when using multiple

GPUs in Figures 6 and 7, which shows the execution time of the kernel was 10,177ns and 11,104ns. When comparing these results, it is clear that a single GPU with a single stream has significantly worse performance when compared to the single GPU with concurrent streams and multiple GPU results. The single GPU with concurrent streams and multiple GPU code has about a 100x performance speed up when compared to the code with a single GPU with a single stream. It can be seen in the profiler output that the single GPU with concurrent streams and the multiple GPU both have 2 instances of kernel calls, meaning that there were 2 streams created and there were 2 GPUs used. The significant performance improvement when using the hardware setting of single GPU with concurrent streams and multiple GPU makes sense, as concurrent streams allow computation tasks to be done in parallel and utilizing multiple GPUs allows for 2 computations to occur at the same time on 2 different GPUs.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	13,606,168	1	13,606,168.0	13,606,168.0	13,606,168	13,606,168	0.0	8192	1	1	256	1	1	addVectorsInto(float *, float *, float *, int)

Figure 8: Profile for Vector Addition Code with Single GPU with Single Stream

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	3,301,670	2	1,650,835.0	1,650,835.0	57,248	3,244,422	2,253,672.3	32	1	1	256	1	1	addVectorsInto(float *, float *, float *, int)

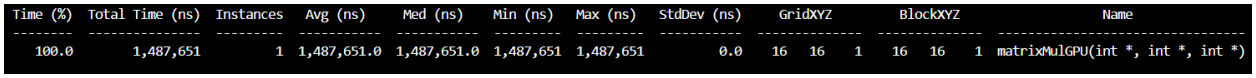
Figure 9: Profile for Vector Addition Code with Single GPU with Concurrent Streams

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			Name
100.0	113,857	2	56,928.5	56,928.5	56,928	56,929	0.7	32	1	1	256	1	1	addVectorsInto(float *, float *, float *, int)

Figure 10: Profile for Vector Addition Code with Multiple GPUs

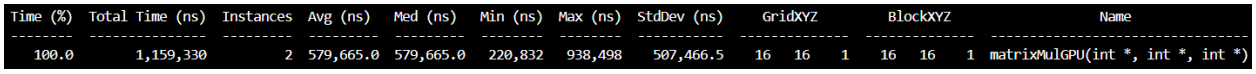
The next code that was analyzed for the different hardware settings was the vector element addition code. The program was initialized to have vectors of 2 << 20 int elements. One can see the profile results for this code when using a single GPU with a single stream in Figure 8, which shows that the execution time of the vector addition kernel took 13,606,168ns. One can also see the profile results for this code when using a single GPU with multiple streams and when using multiple GPUs in Figures 6 and 7, which

shows the execution time of the kernel was 3,301,670ns and 113,857ns. When comparing these results, it is clear that a single GPU with a single stream has significantly worse performance when compared to the single GPU with concurrent streams and multiple GPU results. The single GPU with concurrent streams has about a 4x performance speed up when compared to the code with a single GPU with a single stream. The multiple GPU code has about a 120x performance speed up when compared to the code with a single GPU with a single stream. The hardware setting of a single GPU with concurrent streams provides a viable performance speed up, while the multiple GPU hardware setting provides a significant performance speed up. Again, it can be seen in the profiler output that the single GPU with concurrent streams and the multiple GPU both have 2 instances of kernel calls, meaning that there were 2 streams created and there were 2 GPUs used. The performance improvement when using the hardware setting of single GPU with concurrent streams, as well as the significant improvement when using multiple GPUs makes sense, as concurrent streams allow computation tasks to be done in parallel and utilizing multiple GPUs allows for 2 computations to occur at the same time on 2 different GPUs. The difference between the performances of the single GPU with concurrent streams and the multiple GPU hardware setting arises from the size of the vectors. 2<<20 is equivalent to 2,097,152, which is a very large size for a vector. When performing the kernel execution for concurrent streams, only one stream can utilize the ALU at a time, so while one stream is performing the addition operation, the other stream is performing some memory operation. This parallelism is what provides the performance improvement compared to the single GPU with a single stream hardware setting. When performing the kernel execution for multiple GPUs, since the two GPUs are isolated and there are no data dependencies, each GPU can perform the computation on the vector elements simultaneously. Since the vector is very large, performing the vector addition computations simultaneously on the two GPUs is what provides the large performance improvement compared to the single GPU with a single stream hardware setting, as well as the improvement compared to the single GPU with concurrent streams hardware setting.



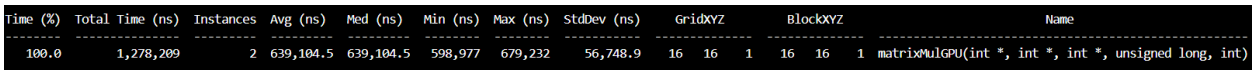
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
100.0	1,487,651	1	1,487,651.0	1,487,651.0	1,487,651	1,487,651	0.0	16 16 1	16 16 1	matrixMulGPU(int *, int *, int *)

Figure 11: Profile for 2D Matrix Multiplication Code with Single GPU with Single Stream



Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
100.0	1,159,330	2	579,665.0	579,665.0	220,832	938,498	507,466.5	16 16 1	16 16 1	matrixMulGPU(int *, int *, int *)

Figure 12: Profile for 2D Matrix Multiplication Code with Single GPU with Concurrent Streams



Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
100.0	1,278,209	2	639,104.5	639,104.5	598,977	679,232	56,748.9	16 16 1	16 16 1	matrixMulGPU(int *, int *, int *, unsigned long, int)

Figure 13: Profile for 2D Matrix Multiplication Code with Multiple GPU

The last code that was analyzed for the different hardware settings was the 2D matrix multiplication code. The program was initialized to have matrices of size 256x256, which corresponds to vectors of size 65,536. One can see the profile results for this code when using a single GPU with a single stream in Figure 11, which shows that the execution time of the matrix multiplication kernel took 1,487,651ns. One can also see the profile results for this code when using a single GPU with multiple streams and when using multiple GPUs in Figures 12 and 13, which shows the execution time of the kernel was 1,159,330ns and 1,278,209ns. When comparing these results, it can be seen that a single GPU with a single stream has worse performance when compared to the single GPU with concurrent streams and multiple GPU results, although it is marginal. This marginal performance improvement can be attributed to the nature of the code. The matrix multiplication code contains data dependencies, so memory management was required to be performed. When working with GPUs, one of the biggest performance inhibitors is memory transfer, as it takes a significant amount of time compared to the actual computations, which is why the performance improvement is not as significant compared to the other codes. All and all, it can be seen that the hardware settings of a single GPU with concurrent streams and multiple GPUs provided performance improvements compared to the default hardware setting of a single GPU with a single stream. Although

the performance improvements are clear, it can also be noted that the optimal hardware setting to use depends on the code and task at hand, as well as available resources such as the number of GPUs available.

Conclusion

In conclusion, this comprehensive analysis of GPU hardware settings reveals significant insights into optimizing performance for computational tasks. Through experimentation with three different hardware configurations of single GPU with single stream, single GPU with concurrent streams, and multiple GPUs across various code implementations, clear trends emerge regarding their impact on execution times. The findings indicate that leveraging concurrent streams and multiple GPUs can substantially enhance performance compared to the default single GPU with a single stream configuration. Concurrent streams enable parallel execution of computational tasks within a single GPU, while multiple GPUs enable simultaneous processing across isolated devices. These configurations prove particularly advantageous for tasks involving high computational demand and large-scale workloads.

Moreover, the observed performance improvements vary across different types of computational tasks. While tasks with minimal data dependencies, such as vector element doubling and addition, exhibit significant speedups with concurrent streams and multiple GPUs, tasks with more complex data dependencies, like 2D matrix multiplication, show more marginal improvements. This underscores the importance of considering both the nature of the task and available resources when selecting the optimal hardware configuration. Additionally, it highlights the necessity of efficient memory management, especially for tasks involving data-intensive operations, to fully capitalize on the potential performance gains offered by advanced hardware configurations.

In essence, this study emphasizes the critical role of GPU hardware settings in optimizing computational performance and underscores the need for tailored approaches based on specific task requirements and

available resources. By understanding and leveraging the capabilities of different hardware configurations, researchers and practitioners can unlock the full potential of GPU-accelerated computing for a wide range of applications.

Appendix A: Double Vector Code Examples

```
cudaMallocManaged(&a, size); // allocated memory

size_t threads_per_block = 1024;  // set threads per block and # of blocks
size_t number_of_blocks = 32;

doubleElements<<<number_of_blocks, threads_per_block>>>(a, N); // call kernel

cudaError_t err11 = cudaGetLastError(); // returns the error from above.
cudaError_t err22 = cudaDeviceSynchronize(); // ensure GPU threads all finish
```

A 1.1: Single GPU Single Stream Code for Double Vector Elements

```
// allocate memory for cpu and gpu data
cudaMallocHost(&data_cpu, size);
cudaMalloc    (&data_gpu, size);

size_t threads_per_block = 1024; // set threads per block and # of blocks
size_t number_of_blocks = 32;

// Set the number of streams to not evenly divide num_entries.
const uint64_t num_streams = 2;

cudaStream_t streams[num_streams]; // create stream array and streams
for (uint64_t stream = 0; stream < num_streams; stream++)
    cudaStreamCreate(&streams[stream]);

// get data per stream
int x = N;
int y = num_streams;
const uint64_t chunk_size = x/y + (x % y != 0);
```

A 2.1: Single GPU Concurrent Streams Double Vector Elements Code for Variable Initialization

```
for (uint64_t stream = 0; stream < num_streams; stream++) {
  // get data width
  const uint64_t lower = chunk_size*stream;
  const uint64_t upper = min(lower+chunk_size, N);
  const uint64_t width = upper-lower;

  // asynchronously copy data from host to device
  cudaMemcpyAsync(data_gpu+lower, data_cpu+lower,
                  sizeof(int)*width, cudaMemcpyHostToDevice,
                  streams[stream]);

  doubleElements<<<number_of_blocks, threads_per_block, 0, streams[stream]>>>(data_gpu+lower, N);

  cudaError_t err11 = cudaGetLastError(); // return the error from above.
  cudaError_t err22 = cudaDeviceSynchronize(); // ensure GPU threads all finish

  // asynchronously copy data from device to host
  cudaMemcpyAsync(data_cpu+lower, data_gpu+lower,
                  sizeof(int)*width, cudaMemcpyDeviceToHost,
                  streams[stream]);
}

// Destroy streams.
for (uint64_t stream = 0; stream < num_streams; stream++)
  cudaStreamDestroy(streams[stream]);
```

A 2.2 : Single GPU Concurrent Streams Double Vector Elements Code for Calling Kernel

```
// get # of GPUs
int num_gpus;
cudaGetDeviceCount(&num_gpus);

size_t threads_per_block = 1024; // set threads per block and # of blocks
size_t number_of_blocks = 32;

// get data per gpu
int x = N;
int y = num_gpus;
const uint64_t chunk_size = x/y + (x % y != 0);

uint64_t *data_gpu[num_gpus]; // One pointer for each GPU.

for (int gpu = 0; gpu < num_gpus; gpu++) {

  cudaSetDevice(gpu); // set gpu to be used

  const uint64_t lower = chunk_size*gpu;
  const uint64_t upper = min(lower+chunk_size, size);
  const uint64_t width = upper-lower; // get data width

  cudaMalloc(&data_gpu[gpu], sizeof(uint64_t)*width); // Allocate chunk of data for current GPU.
}
```

A 3.1: Multiple GPU Double Vector Elements Code for Variable Initialization

```
for (int gpu = 0; gpu < num_gpus; gpu++) {
  cudaSetDevice(gpu); // set gpu to be used

  const uint64_t lower = chunk_size*gpu; // get data width
  const uint64_t upper = min(lower+chunk_size, N);
  const uint64_t width = upper-lower;

  cudaMemcpy(data_gpu[gpu], a+lower, // copy data from host to device
      sizeof(uint64_t)*width, cudaMemcpyHostToDevice);
}
```

A 3.2: Multiple GPU Double Vector Elements Code for Variable Assignment

```
for (int gpu = 0; gpu < num_gpus; gpu++) {
  cudaSetDevice(gpu); // get data per gpu

  const uint64_t lower = chunk_size*gpu; // get data width
  const uint64_t upper = min(lower+chunk_size, N);
  const uint64_t width = upper-lower;

  // Pass chunk of data for current GPU to work on.
  doubleElements<<<number_of_blocks, threads_per_block>>>(data_gpu[gpu], width);

  cudaError_t err11 = cudaGetLastError(); // return the error from above.
  cudaError_t err22 = cudaDeviceSynchronize(); // ensure GPU threads all finish
}
```

A 3.3: Multiple GPU Double Vector Elements Code for Calling Kernel

```
for (int gpu = 0; gpu < num_gpus; gpu++) {
  cudaSetDevice(gpu); // set gpu to be used

  const uint64_t lower = chunk_size*gpu; // get data width
  const uint64_t upper = min(lower+chunk_size, N);
  const uint64_t width = upper-lower;

  cudaMemcpy(a+lower, data_gpu[gpu], // copy data from device to host
    sizeof(uint64_t)*width, cudaMemcpyDeviceToHost);
}
```

A 3.4: Multiple GPU Double Vector Elements Code for Memory Transfer

References

[1] Cabrera, Fang *The CUDA Parallel Programming Model - 9. Interleave Operations by Stream* (2019) [Figure] https://nichijou.co/cuda9-stream2/

[2] ResearchGate *Multi-Stream MPS and MIG Illustration* [Figure] https://www.researchgate.net/figure/Multi-Stream-MPS-and-MIG-Illustration-ture-Figure-2-a-As-a-software-based_fig1_359309310

[3] NVIDIA *Fundamentals of Accelerated Computing with CUDA C/C++* (2023) https://learn.nvidia.com/courses/course?course_id=course-v1:DLI+C-AC-01+V1&unit=block-v1:DLI+C-AC-01+V1+type@vertical+block@b8bf5b6c2f3b4697a2f0751fc4749643

[4] NVIDIA *Accelerating CUDA C++ Applications with Concurrent Streams* (2023) https://learn.nvidia.com/courses/course-detail?course_id=course-v1:DLI+S-AC-01+V1

[5] NVIDIA *Scaling Workloads Across Multiple GPUs with Cuda C/C++* (2023) https://learn.nvidia.com/courses/course-detail?course_id=course-v1:DLI+S-AC-02+V1