NASA/TM-20240013384

# Runtime Thread-Block Optimization for Custom Multistream CUDA Kernels for Glenn Research Center Communication Analysis Suite

*Aden Bergstresser and Bryan W. Welch*
*Glenn Research Center, Cleveland, Ohio*

December 2024

# NASA STI Program Report Series

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION.
  Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM.
  Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT.
  Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION.
  Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION.
  Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION.
  English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

NASA/TM-20240013384

# Runtime Thread-Block Optimization for Custom Multistream CUDA Kernels for Glenn Research Center Communication Analysis Suite

*Aden Bergstresser and Bryan W. Welch*
*Glenn Research Center, Cleveland, Ohio*

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

December 2024

This report contains preliminary findings, subject to revision as analysis proceeds.



*Level of Review*: This material has been technically reviewed by technical management.

This report is available in electronic form at https://www.sti.nasa.gov/ and https://ntrs.nasa.gov/

NASA STI Program/Mail Stop 050
NASA Langley Research Center
Hampton, VA 23681-2199

# Runtime Thread-Block Optimization for Custom Multistream CUDA Kernels for Glenn Research Center Communication Analysis Suite

Aden Bergstresser* and Bryan W. Welch
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

## Summary

In preparation for the upcoming Artemis missions and the return of humans to the Moon, NASA scientists must evaluate proposed landing sites for terrain and communications viability. The Glenn Research Center Communication Analysis Suite (GCAS) combines sophisticated communication network models with accurate lunar terrain to access sites across the Moon's south pole. Given the importance of proper site selection to crew safety and mission success, many locations must be analyzed, resulting in the need to perform a large computational load. To meet the growing project demands, development has begun to improve the runtime efficiency of GCAS with graphics processing unit (GPU) parallelization by way of Compute Unified Device Architecture (CUDA, NVIDIA's proprietary parallel computing API) multistream kernels.

One of the most prominent factors in kernel optimization is the proper selection of thread-block dimensions to maximize the concurrent operation on the device. Typically, thread-block dimensions are optimized by hand, requiring many stages of benchmarking and iteration. Additionally, given the main conditions to optimization are the physical GPU architecture and problem size, these optimal dimensions are nonportable and fragile in their scope. As such, a novel optimization routine was developed to generate the optimal thread-block dimensions during runtime with considerations for hardware specifications and problem size, resolving the issues of portability and enabling the function of more dynamic routines.

## Nomenclature

| | |
|---|---|
| / | standard decimal division |
| // | integer division |
| 1D | one-dimensional |
| 2D | two-dimensional |
| 3D | three-dimensional |
| Block | group of threads |
| CCX.Y | Compute Capability Generation X.Y |
| CPU | central processing unit |
| CUDA | Compute Unified Device Architecture, NVIDIA's proprietary parallel computing API |
| D2H | device-to-host |
| Device | GPU |
| GCAS | Glenn Research Center Communication Analysis Suite |
| GEMV | General Matrix-Vector Multiplication |

*NASA Office of STEM Engagement Summer 2024 Intern, University of Michigan undergraduate.

| | |
|---|---|
| GPU | graphics processing unit |
| Grid | group of blocks |
| H2D | host-to-device |
| HDF5 | Hierarchical Data Format Version 5 |
| Host | CPU |
| LOS | line of sight |
| SIMT | single instruction, multiple threads |
| SM | streaming multiprocessor |
| Stream | Single operation sequence |
| Thread | Single kernel operating entity |
| Warp | Group of threads that execute in SIMT |

# 1.0 Introduction

The terrain-generation capabilities of the Glenn Research Center Communication Analysis Suite (GCAS) focus on taking in lunar surface geometry data in the form of Hierarchical Data Format Version 5 (HDF5) files and constructing a topological mask of the terrain to be used in line-of-sight (LOS) and diffraction communication link analysis. Due to the high level of detail of the satellite data in conjunction with the large surface areas being analyzed, the terrain mask generation routines are very resource and time intensive. Given that they are largely ubiquitous within the larger communication analysis functions, improvements in both computational efficiency and program runtime are strongly desired to increase the analysis throughput GCAS can support.

Subroutines in this task are largely divided into two main groups: element-wise matrix computations and data partitioning and restructuring. For both actions, the operation of individual data elements is independent of those in the rest of the set, making them both prime candidates for parallelization. This led to the decision to provide graphics processing unit (GPU) parallelization functionality by way of CUDA to the GCAS framework.

To outperform the current C++ implementations that utilize the Eigen library for element-wise matrix operations and the C++ standard library for efficient memory management, extensive optimization techniques were necessary in the new CUDA framework. The primary techniques chosen to implement that are discussed in this report were thread-block dimension optimization at runtime and Multiple Asynchronous Nondefault Stream usage.

# 2.0 Thread Hierarchy Summary

CUDA parallelization uses the single instruction, multiple threads (SIMT) model for its multithreaded executions (Ref. 1); therefore, each thread handles a single execution of the operating kernel, with all of them operating simultaneously. Note that here, “kernel” does not refer to the operating system kernel on the central processing unit (CPU); instead, it is associated with the GPU.

For each kernel execution, threads are allocated in batches according to the CUDA thread hierarchy illustrated in Figure 1. In this model, each thread performs a kernel operation on one element of the input data. Then these threads are grouped together in thread-blocks, which all operate together on a single multiprocessor with shared memory between them. Finally, all of the blocks are grouped into a grid that is allocated to a single device. Thus, the total number of threads launched and, by extension, operations performed is equivalent to the product of the block size with the grid size (Ref. 1).

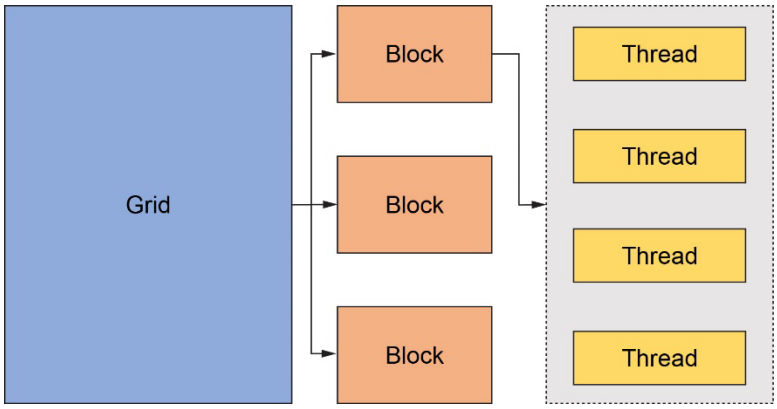

Figure 1.—CUDA thread hierarchy. Grids are composed of blocks and blocks are composed of threads.

### 2.1 Thread-Block Dimensions

Among these three thread groupings, determining the appropriate block dimensions is the most important in terms of runtime optimization because using the device's streaming multiprocessors (SMs) efficiently is paramount.

The primary factors impacting the optimal size of the thread-blocks are the kernel being executed, the physical device architecture being used, and the size of the problem being operated on.

Methods for determining the impacts of each of these factors are discussed in detail within the thread-block optimization routines.

## 3.0 Asynchronous Multistream Summary

A stream in CUDA is simply a sequence of sequential operations that operate on the GPU. These streams allow for discrete portions of a problem to be executed concurrently, greatly improving runtime efficiency because of increased parallelization.

By using asynchronous memory copy methods like cudaMemcpyAsync, individual streams can be overlapped, where loading data from the CPU to GPU on one part of the data can occur at the same time that a kernel is operating on another part, while yet another portion of the data that was already operated on can be moved back from the GPU to the CPU. These asynchronous methods avoid blocking that could occur using synchronous methods, enabling this concurrency. Additionally, exclusively nondefault streams are utilized because the default stream only allows for synchronous transfers, resulting in bottlenecks for function execution (Ref. 2).

Methods for implementing the use of multiple asynchronous streams are discussed in detail in Section 5.0 of this report, "Multistream Functionality Implementation."

## 4.0 Thread-Block Optimization Routine

To resolve the issues of kernel portability to different hardware and problem sizes, a runtime thread-block optimization routine based on the work of Mukunoki, Imamura, and Takahashi (Ref. 2) was implemented in the code base.

## 4.1 Retrieve Device Parameters

Before discussing the implementation of the runtime thread-block optimization routine, it was necessary to retrieve the specifications on the GPU hardware being used and the kernel being executed. To get the device parameters, Algorithm 1 was created, which used a combination of cudaDeviceProps methods and hard-coded selection criteria to get Compute Capability Generation-specific information. Because the architecture of the GPU is static at runtime, this routine is performed a single time asynchronously at the top of the GCAS routine, where the cudaDeviceProps struct is stored for the lifetime of the analysis instance to reduce individual kernel overhead, improving efficiency.

The output of this function would return a new struct labeled "deviceProps," which included the fields shown in Figure 2, to be used in the block optimization routines. Table I lists the field labels and their meanings.

```
GetDeviceProps:
Input: deviceID
Output: outputDeviceProps
{
cudaGetDevice(&deviceId);
cudaGetDeviceProperties(&deviceProps, deviceID);

outputDeviceProps.ws = deviceProps.warpSize;
outputDeviceProps.dpMultiprocessorCount = deviceProps.multiProcessorCount;
outputDeviceProps.dpRegsPerMulitprocessor = deviceProps.regsPerBlock;
outputDeviceProps.shmcap = deviceProps.sharedMemPerBlock;

# COMMENT:  if conditions simplified for brevity where the completely capitalized variables are
#                 hardcoded values from Nvidia documentation

if (deviceProps.major == x && deviceProps.minor = y) do
            outputDeviceProps.raus = RAUS;
            outputDeviceProps.wag = WAG;
            outputDeviceProps.wMax = WMAX;
            outputDeviceProps.bMax = BMAX;
end if

return outputDeviceProps;
}
```

Algorithm 1.—Retrieves all relevant device specifications, taking deviceID as input and returning a cudaDeviceProps struct containing requested information.

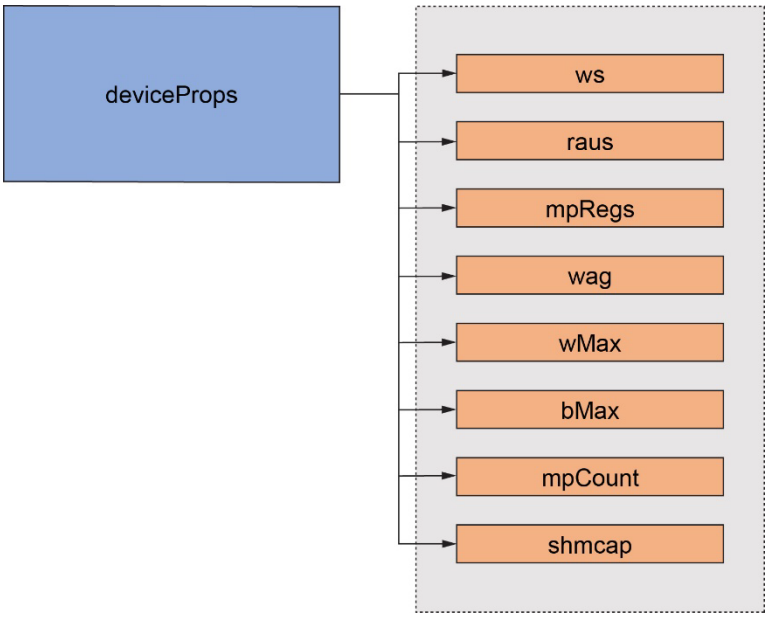


Figure 2.—Outline of deviceProps struct.

TABLE I.—deviceProps FIELD LABELS AND MEANINGS

| *Field* | **Meaning** |
|---|---|
| *ws* | Warp size |
| *raus* | Register allocation unit size |
| *mpRegs* | Number of Registers per SM |
| *wag* | Warp allocation granularity |
| *wMax* | Maximum number of active warps per SM |
| *bMax* | Maximum number of active blocks per SM |
| *mpCount* | Number of SMs on the device |
| *shmcap* | Maximum amount of shared memory on the device |

## 4.2 Retrieve Function Attributes

A method similar to the getDeviceProps function already described had to be implemented to retrieve information on the kernel attributes to properly optimize the block size for the kernel in question. This method would be inserted into the kernel wrapper just prior to running the optimization method described in “GPU Occupancies.” Algorithm 2 outlines the method used to achieve this using the cudaFuncAttributes methods (Ref. 1).

## 4.3 GPU Occupancies

The block optimization method described in detail in this subsection effectively checks what block size results in the greatest device occupancies, which it defines as the most optimal size. These occupancies are separated into three types: warp, block, and grid. They are determined using the getOcps() method from Mukunoki, Imamura, and Takahashi (Ref. 3).

```
GetKernelAttributes:
Input: kernel
Output: numRegs, numThreads
{
cudaError_t buffer = cudaFuncGetAttributes(cudaFuncAttributes*, kernel);
numRegs = cudaFuncAttributes.numRegs;
numThreads = cudaFuncAttributes.maxThreadsPerBlock;

return {numRegs, numThreads};
}
```

Algorithm 2.—Routine to get function attributes of a kernel, taking kernel as input and returning number of registers and threads per block that kernel can access.

Warp occupancy is a measure of warp-level parallelism and is defined as the percentage of active warps compared to the maximum number of allowed warps on an SM.

Block occupancy is a measure of the thread concurrency on a given SM as defined as the percentage of active threads against the maximum number of threads per SM.

Grid occupancy is discussed in detail in "Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs" also by Mukunoki, Imamura, and Takahashi (Ref. 4), where it is referred to as "block occupancy." Grid occupancy is a measure of SM usage across the device and is dependent on the problem size and kernel, unlike the other two occupancies.

For ease of use within the code base, these three occupancies were compiled into a single struct called gpuOccupancies with fields wo, bo, and go for warp, block, and grid occupancy, respectively.

## 4.4 Generate Optimal Dimensions

To implement the thread-block optimization for the GCAS use case, a modification of Algorithm 2 from Mukunoki, Imamura, and Takahashi (Ref. 3) was created that altered the two-dimensional (2D) block-dimension allocation routine to be one-dimensional (1D), Algorithm 3, which is more in line with the current C++ and original MATLAB® (The MathWorks, Inc.) routines to aid in traceability within the project.

One drawback of using a 1D indexing method rather than 2D is that it limits the total number of threads that can be launched. Block and grid dimensions are defined by dim3 data types and so are composed of three nonzero positive integers that specify the size of the structure. The maximum X dimension on modern CUDA GPU architecture is 2,147,483,647 threads, which means that no more than that number of threads can be launched by a single kernel; meanwhile, with 2D indexing, given that the maximum Y dimension is 65,535 threads, far more threads can be launched. Of course, this principle extends to three-dimensional (3D) indexing as well, with the maximum Z dimension also being 65,535 threads.

```
GetKernelAttributes:
Input: deviceProps, problemSize, numRegs, numBlocks, shm
Output: optBlockDims
{

# COMMENT: suffOcps is sufficient occupancies which is determined using the preprocessing routines
#                described in detail later in the paper

minX = 1;
maxX = numBlocks
bestOcps = getOcps(deviceProps, numRegs, maxX, shm, problemSize);
while maxX >= minX do
        if maxX % deviceProps.ws == 0 do
                currOcps = getOcps(deviceProps, numRegs, maxX, shm, problemSize);
                if currOcps >= bestOcps || currOcps >= suffOcps do
                        bestOcps = currOcps;
                        blockSize = maxX;
                end if
        end if
        --maxX;
end while
dim3 optBlockDims(blockSize);

return optBlockDims;
}
```

Algorithm 3.—1D thread-block optimization routine taking device properties, problem size, kernel properties, and amount of shared memory as input and returning a dim3 type (a structure designed for storing dimensions of blocks and grids) containing optimal thread-block dimensions.

This drawback is insignificant in the vast majority of use cases given that the number of concurrent thread executions is far lower, resulting in no runtime benefit from these additional thread launches; due to launch overhead, these excess kernels would likely impede the kernel operation.

# 5.0 Multistream Functionality Implementation

In addition to the thread-block optimization described in the previous section, the use of multiple nondefault streams was implemented to allow for parallelization between data transfers and kernel execution, resulting in greater GPU utilization and lower concurrent memory loads on the device.

## 5.1 Stream Dimension Determination

After running the thread-block optimizations, first, the optimal number of streams would be determined using Algorithm 4. Taking the problem size as input, the routine attempts to optimize the amount of stream concurrency by taking the modulus of the problem size with the integers 2 to 16

defining the number of streams to be launched and, by extension, the amount of concurrent kernel execution. By taking the modulus, the routine checks the implied unused memory created by that number of stream allocations and attempts to minimize this value to optimize the percentage of working stream memory. The routine counts down from 16 to 2 so that if multiple streams with complete memory overlap (i.e., the modulus is zero) are found, then the greater number of streams are selected to increase the amount of concurrent kernel execution.

The range of 2 to 16 was chosen to set the maximum and minimum bounds for concurrent kernel execution while still enabling multiple instances of GCAS to be run concurrently. As of Compute Capability Generation 7.5 (CC7.5), a single device can handle a maximum of 128 concurrent kernel executions; as such, 16 is not necessarily the maximum number of streams that could be launched in theory. However, given that the standard analysis process of GCAS requires multiple instances of the software running concurrently, setting the maximum streams higher than 16 quickly uses up all device resources on single GPU systems, thus failing to meet the requirement of supporting multiple instances. As discussed in Section 8.0, "Topics for Future Awareness," this artificial limit on kernel concurrency can be lifted on multi-GPU systems once multi-GPU support is implemented in the kernel wrappers.

```
GetNumStreams:
Input: problemSize
Output: numStreams
{
smallestRemainder = problemSize;
count = 16;
while numStreams  >= 2 do
        remainder = problemSize / numStreams;
        if remainder == 0 do
                return numStreams;
        end if
        else if remainder < smallestRemainder do
                newSmallest = numStreams;
                smallestRemainder = remainder;
        end else if
end while
return newSmallest;
}
```

Algorithm 4.—Function to return optimal number of streams taking problem size as input.

```
GetStreamSize:
Input: problemSize, numStreams
Output: streamSize
{
if problemSize % numStreams == 0 do
        return problemSize / numStreams;
end if
return ceil (problemSize / numStreams);
}
```

Algorithm 5.—Function to calculate optimal stream size, taking problem size and number of streams as input.

After determining the optimal number of streams to be created, the size of each stream is generated using Algorithm 5. Taking the problem size and number of streams as input, the function returns the optimal stream size in units of data elements. As mentioned previously with Algorithm 4, if problem size is a multiple of the number of streams, there is a complete one-to-one mapping of data to space in the stream; thus, the stream size returned would simply be the quotient of the problem size with respect to the number of streams. If this condition is not met, however, a single element ceiling operation is performed to ensure that all the problem data can be mapped to a stream with minimal overlap. At most, this overlap would be one less data chunk than the maximum number of concurrent kernel executions described in the number of streams generation routine. These data chunks are the groups of input and output elements that need to be allocated to the device; for example, adding two integer vectors and outputting a third would have data chunks of the size of three integers. Therefore, depending on the kernel being run, this memory allotment can be nonnegligible, which further highlights the importance of launching the appropriate number of streams.

The final multistream dimension determination function is the generate optimal grid-size routine. Because kernel executions are staged and run on distinct chunks of the input data, the total number of threads launched per kernel is determined by a ceiling operation performed on the quotient of stream size in place of the problem size with the optimal block size determined using Algorithm 5. Again, it should be highlighted that any residual rounding from the prior steps is compounded into excess resource use; as in this case, more threads could be launched than necessary, but due to the aforementioned optimizations in most circumstances, this excess would be minimal.

## 5.2 Multistream Thread Indexing

Due to the thread-block optimizations utilizing the full problem size in place of the stream size, indexing threads within each kernel execution needs to be incremented by an offset according to the portion of the data it is operating on.

```
Index = threadIdx.x + blockIdx.x * blockDim.x;
```

Code snippet 1.—Standard 1D thread indexing method.

```
Index = offset + threadIdx.x + blockIdx.x * blockDim.x;
```

Code snippet 2.—Augmented 1D thread indexing method for multistream support.

A typical 1D thread indexing method may look something like Code snippet 1, where all the indexing within a kernel is with respect to a single block. However, due to the partitioning of the work across multiple streams, an offset needs to be introduced where the indexing is shown in Code snippet 2.

The proper offset is quite trivial to determine and is simply defined as the product of the specific stream number with the stream size.

## 6.0 CUDA Kernel Wrapper Implementation

To preserve a single functional interface within the code, C++ wrappers were made for each kernel, streamlining development and traceability within the code base. The wrappers were divided into four main parts: device memory allocation, thread hierarchy optimization, host-to-device (H2D) transfers, and device-to-host (D2H) transfer for the function output. An example wrapper is shown in Algorithm 6.

For each vector or matrix input to the function, a corresponding C-style array had to be generated and have its memory allocated on the device due to the choice not to use shared-memory resources. This decision to avoid using shared memory was initially made as an attempt to reduce CPU usage, enabling more instances of GCAS to be run concurrently, increasing analysis throughput. Additionally, limiting the host usage with shared memory makes potential host-device parallelism more achievable for future development.

After allocating all device space, the thread-block optimization routine in Code snippet 2 would be run with the total problem size being determined by the size of the input data.

Then, using the asynchronous structure (Ref. 5) illustrated in Figure 3, H2D transfers are first looped over allocating portions of the input data to each stream as described previously in this section, then looping over kernel executions with the block size determined by the thread-block optimization with the grid size determined during the stream dimensionality routines, such that the number of total thread launches per stream was enough to minimally cover the data contained in each stream. Finally, D2H transfers are looped over, placing the data directly into the output container on the CPU, using the data() method for std and Eigen vectors to decompose them into equivalent C-style arrays. This would result in a zero-copy host transfer, improving memory and runtime efficiencies.

```
ExampleWrapper:
Input: vec_a, vec_b
Output: vec_c
{
d_a*, d_b*, d_c*;
problemSize = vec_a.size();
GetKernelAttributes();
threadBlockOpt(problemSize);
cudaMalloc(d_a, problemSize);
cudaMalloc(d_b, problemSize);

cudaStream_t streams[numStreams];

for (stream : streams) do
        cudaStreamCreate(stream);
end for

for (stream : streams) do
        hostToDevTransfer(stream);
end for

for (stream : streams) do
        kernelExecution();
end for

for (stream : streams) do
        devToHostTransfer();
end for

for (stream : streams) do
        cudaStreamDestroy(stream);
end for

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return vec_c;
}
```

Algorithm 6.—Example wrapper taking two vectors as input and one vector as output.

Note that this level of parallelism is not enabled on all device architectures due to the requirement for distinct H2D and D2H channels. Most modern GPUs have support for these dual channels, which is why this architecture was highlighted. Also note that for all GPUs after CC3.5, the Hyper-Q feature, which enables multiple CPU threads or processes to launch work on a single GPU simultaneously, eliminates the need for controlled launch order. Thus, the asynchronous V1 and V2 methods described in Reference 5 perform equivalently.

As a result of overlapping these data transfers using concurrent stream execution, a significant improvement in runtime is observed due to the increased parallelization of the routine. Figure 4 shows a comparison in runtime efficiency of kernel execution with and without multistream functionality, where it is assumed for clarity that all data transfers and kernel executions take the same amount of time, and while the speed of these operations was independent of any changes between single and multistream usage.

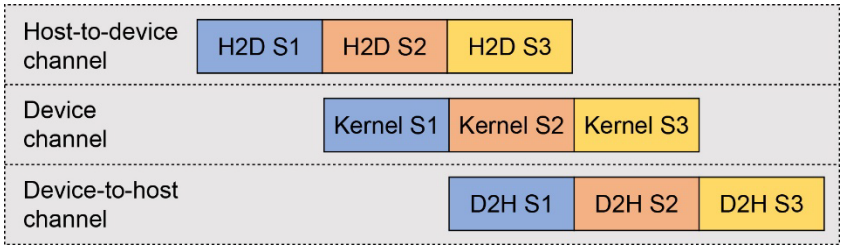


Figure 3.—Stream execution diagram for asynchronous structure of overlapping data transfers (Ref. 5).

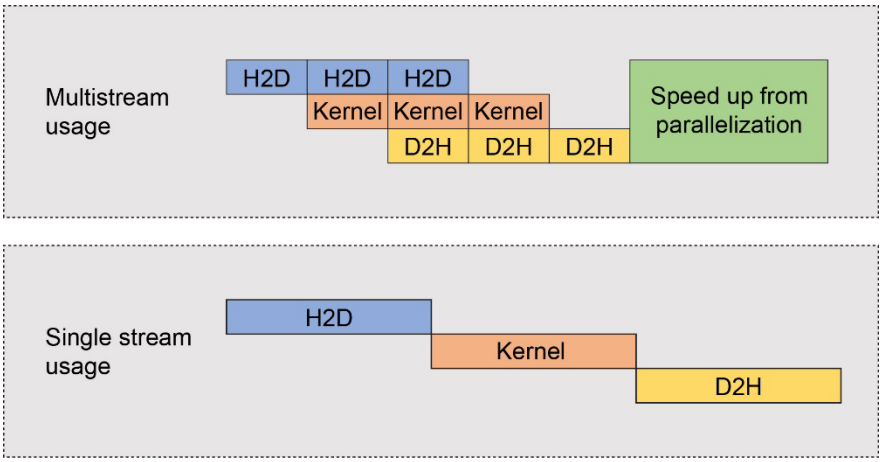


Figure 4.—Comparison of single stream versus multistream runtimes.

# 7.0 Sufficient Occupancies Generation Routine

To determine the sufficient occupancies for the thread-block optimization functions, an automatic preprocessing routine was implemented that profiled the kernel runtime while recording the device occupancies for every allowed block dimension combination.

Using these results, this routine determines the appropriate sufficient occupancies to be used and writes them to an HDF5 file (Ref. 6), which are read to implement these occupancies automatically into the optimization routines.

Because this routine is implemented using kernel benchmarking rather than with strict models as was done with the block and stream optimizations discussed in Section 6.0, these sufficient occupancies are not necessarily portable between different kernels or device architectures. This is a clear limitation of this routine.

## 7.1 Profile Step Generation

To profile the different kernel launches, a struct called profileStep was created with member types of gpuOccupancies and std::chrono::duration<long long, std::nano>. The gpuOccupancies is the same struct that was used in the thread-block optimizations; std::chrono::duration<long long, std::nano> is a type from the chrono library in C++. To profile the kernel for each possible block size, a method very similar to Algorithm 1 was used, in which the code block contained within the first if-statement inside the while-loop was adjusted to collect these profileStep objects.

Code block 1 is used to first calculate the GPU occupancies resulting from the thread-block size being profiled at that particular step, then a std::chrono::high_resolution_clock is started to mark the beginning of the kernel launch. This clock is immediately followed by the preprocessing kernel wrapper described in detail in Code block 1, which in turn is followed by a second clock being started to mark the end of the kernel execution. Using std::chrono::duration allows extracting the runtime of the kernel launch with nanosecond accuracy. This duration, along with the occupancies, is then inserted into a profileStep object, which is appended to a vector of profileSteps containing the results from each block size profile.

```
Input: currTestBlockSize
Output: profileSteps (std::vector<profileStep>)

gpuOccupancies currOcc = getOcc();
dim3 testBlock(currTestBlockSize);
start = std::chrono::high_resolution_clock::now();
preprocessingKernelWrapper(testBlock);
finish = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration(finish – start);
tempProfileStep.occ = currOcc;
tempProfileStep.dur = duration;
profileSteps.push_back(tempProfileStep);
```

Code block 1.—profileSteps generation routine.

## 7.2 Determine Sufficient Occupancies

After collecting all the profileStep data using the methods already described, that data is passed to Algorithm 7, which determines the correct sufficient occupancies to be used by block optimization functions. This function returns the sufficient occupancies by inspecting a subset of the collected profiling determined by the top cutoffPercent of times. For example, if cutoffPercent was 10, then only the top 10 percent of fastest kernel runtimes in profileSteps would be looked at. This subset would then be looped over to find the minimum occupancy level for each of the three occupancy types: warp, block, and grid. These values are minimized to lessen the constraints on optimization as much as possible in order to ensure that even odd-sized problems can be optimized effectively. In this context, "odd-sized" refers to problem sizes that are not multiples of many of the important optimization criteria like warp size. These odd problem sizes are typically prime or are simply odd in the sense they are not multiples of 2.

```
GetSufficientOccupancies
Input: profileSteps, cutoffPercent
Output: minOccupancy
{
topNumber = profileSteps.size() // cutoffPercent;
topSteps(topNumber);
std::partial_sort_copy(profileSteps.begin(),
                       profileSteps.end(), topSteps.begin(),
                       topSteps.end(), profileStepsComp);
minOccupancy.wo = 0;
minOccupancy.bo = 0;
minOccupancy.go = 0;

for step : topSteps do
        if step.occupancy.wo < minOccupancy.wo do
                minOccupancy.wo = step.occupancy.wo;
        end if
        if step.occupancy.bo < minOccupancy.bo do
                minOccupancy.bo = step.occupancy.bo;
        end if
        if step.occupancy.go < minOccupancy.go do
                minOccupancy.go = step.occupancy.go;
        end if
end for

return minOccupancy;
}
```

Algorithm 7.—Generate sufficient occupancies; profileStepsComp is a simple comparator that returns true when runtime of profileStep A is less than runtime of profieStep B.

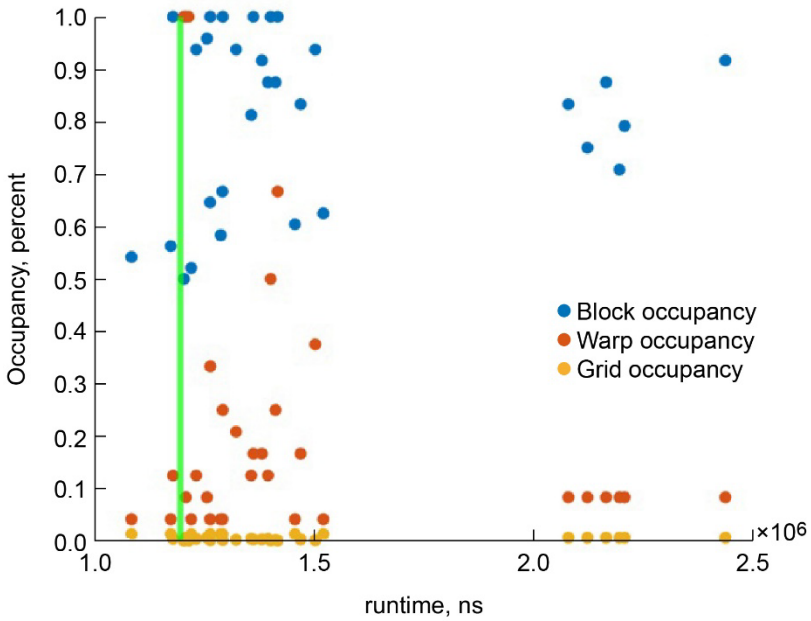


Figure 5.—Profile steps generated from running haversine distance kernel on $2^{20}$ element vectors; solid green line indicates cutoff percent of times as determined by Algorithm 6, which was 10 percent in this example.

### 7.3 Impacts on Haversine Distance Calculation

To test the impact of the preprocessing of the sufficient occupancies on calculating haversine distances, the kernel was first run with manual thread-block optimization, then with the runtime block optimizations described in Section 4.0, and finally with the runtime block optimizations using the sufficient occupancies generated from the preprocessing.

The determination of these sufficient occupancies using the preprocessing routines can be observed in Figure 5.

With manual optimization, the kernel running on all 320 lunar groups took an average of 12 ms to complete; after implementing the thread-block optimization routines, it completed in just 4 ms. This improvement was further amplified by using the sufficient occupancies generated from the preprocessing routines, resulting in a total runtime of just 1 ms, a 12 time improvement in runtime efficiency over manual optimization.

## 8.0 Topics for Future Awareness

Unlike C++ code, CUDA programs are incredibly dependent on the system architecture they are being executed on, resulting in challenges that are unique to CUDA development. This section outlines several topics of interest for future development and limitations of the current implementations.

### 8.1 Shared Memory

Shared memory is an incredibly useful tool for reducing the overhead generated by H2D and D2H transfers because the physical memory location is shared between the device and the host. However, this sharing presents unique challenges in implementation using both CPU and GPU processes, which have to be analyzed to determine if shared memory would be useful and how much of that task should be stored in that memory location.

As such, the current implementations of the functional wrappers neglect the use of shared memory, choosing instead to use pinned memory on both the host and device and transferring that data across the system. The optimization routines, however, do support optimization, with shared memory use greatly easing the burden of development into this area.

### 8.2 Multiple GPU Systems

Another candidate for further optimization would be through the use of multi-GPU systems. By having more SMs provided by the additional devices, kernels can make use of greater concurrency, resulting in improved runtime efficiency.

Much like shared memory, the current optimization routines support multi-GPU usage by looping over each device and performing the optimizations independently of each another. Additionally, minimal changes would be required to the functional wrappers to enable this functionality; simply altering the device ID with a loop would suffice in most circumstances.

However, depending on the process being implemented, changes to the operating kernel may be necessary. For example, with element-wise matrix operations, a single kernel could be used as there is no information that is unique to a single portion of the data. With partitioning routines, however, this is often not the case; different groups of the data are evaluated differently by the partitioning criteria due to group-specific constants, etc.

### 8.3 Compute Capability Generation

The CUDA API currently provides no methods for determining certain Compute Capability Generation-specific limitations such as warp allocation granularity. Thus, these values must be hard-coded into the getDeviceParameters routines; the result of this is that these values must be updated by hand using NVIDIA documentation for guidance whenever a new architecture is released. These generation specifications are easily found using the NVIDIA CUDA Programming Guide (Ref. 7) and NVIDIA GPU consumer documentation (Ref. 8).

As of summer 2024, the GCAS CUDA code can handle optimization for systems ranging from CC2.0 to CC9.0, but this will need to be adjusted in the future; the new Blackwell architecture line of GPUs is set for release later in 2024, bringing CC10.0 with it.

## 9.0 Conclusions

As a result of these improvements, a 16 percent reduction in total runtime of the terrain mask generation routines and a 20 percent reduction in peak central processing unit (CPU) usage over the same routine were observed. These results demonstrate the usefulness of continued Compute Unified Device Architecture (CUDA) development as part of the Glenn Research Center Communication Analysis Suite (GCAS) code conversion project in improving runtime and resource efficiency.

The implementation of the optimization routines addressed in this report will dramatically increase the rate of future CUDA development within the project by eliminating the time-consuming requirement of profiling and hand optimizing each kernel. Additionally, previously forbidden optimization processes, such as data partitioning, have now been enabled and the issues of hardware portability have been resolved.

CUDA is an incredibly promising tool for further optimizing the efficiency of the GCAS code base, allowing for greater amounts of analysis to be performed and improving the assistance it can provide, not only to the Artemis missions but to the many other projects that utilize the software.

## References


1. NVIDIA Corporation: Your GPU Compute Capability. NVIDIA Developer, 2024. https://developer.nvidia.com/cuda-gpus Accessed Nov. 13, 2024.
2. Rennich, Steve: CUDA C/C++ Streams and Concurrency. NVIDIA, 2024. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf Accessed Nov. 13, 2024.
3. Mukunoki, Daichi; Imamura, Toshiyuki; and Takahashi, Daisuke: Automatic Thread-Block Size Adjustment for Memory-Bound BLAS Kernels on GPUs. Presented at the 2016 IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, 2016, pp. 377–384.
4. Mukunoki, Daichi; Imamura, Toshiyuki; and Takahashi, Daisuke: Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs. Presented at the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015, pp. 642–650.
5. Harris, Mark: How to Overlap Data Transfers in CUDA C/C++. NVIDIA Developer Technical Blog, 2012. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/ Accessed Nov. 13, 2024.
6. The HDF Group: HDF5 User Guide. Release 1.14.5, 2024. https://support.hdfgroup.org/documentation/hdf5/latest/_u_g.html Accessed Nov. 13, 2024.
7. NVIDIA Corporation: CUDA C Programming Guide. 2018. https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf Accessed Nov. 13, 2024.
8. NVIDIA Corporation: cudaFuncAttributes Struct Reference. NVIDIA CUDA Library Documentation 3.2 Beta, 2024. https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/structcudaFuncAttributes.html Accessed Nov. 13, 2024.