Accelerating IceCube's Photon Propagation Code with CUDA

Hendrik Schwanekamp¹ · Ramona Hohl¹ · Dmitry Chirkin² · Tom Gibbs¹ · Alexander Harnisch^{3,4} · Claudio Kopper^{3,4} · Peter Messmer¹ · Vishal Mehta¹ · Alexander Olivas⁵ · Benedikt Riedel² · Martin Rongen⁶ · David Schultz² · Jakob van Santen⁷

Received: 5 July 2021 / Accepted: 13 October 2021 / Published online: 9 February 2022
© The Author(s) 2022

Abstract

The IceCube Neutrino Observatory is a cubic kilometer neutrino detector located at the geographic South Pole designed to detect high-energy astrophysical neutrinos. To thoroughly understand the detected neutrinos and their properties, the detector response to signal and background has to be modeled using Monte Carlo techniques. An integral part of these studies are the optical properties of the ice the observatory is built into. The simulated propagation of individual photons from particles produced by neutrino interactions in the ice can be greatly accelerated using graphics processing units (GPUs). In this paper, we (a collaboration between NVIDIA and IceCube) reduced the propagation time per photon by a factor of up to 3 on the same GPU. We achieved this by porting the OpenCL parts of the program to CUDA and optimizing the performance. This involved careful analysis and multiple changes to the algorithm. We also ported the code to NVIDIA OptiX to handle the collision detection. The hand-tuned CUDA algorithm turned out to be faster than OptiX. It exploits detector geometry and only a small fraction of photons ever travel close to one of the detectors.

Keywords GPU · CUDA · Neutrino astrophysics · Ray-tracing · OpenCL

Introduction

The IceCube Neutrino Observatory [1] is the world's premier facility to detect neutrinos with energies above 1 TeV and an essential part of the global multi-messenger astrophysics (MMA) effort. IceCube is composed of 5160 digital optical modules (DOMs) buried deep in glacial ice at the geographical South Pole. The DOMs can detect individual photons and their arrival time with $O(ns)$ accuracy. IceCube has three components: the main array, which has already been described; DeepCore: a dense and small sub-detector that extends sensitivity to ~ 10 GeV; and IceTop: a surface air shower array that studies $O(PeV)$ cosmic rays.

IceCube is a remarkably versatile instrument addressing multiple disciplines, including astrophysics [2–4], the search for dark matter [5], cosmic rays [6], particle physics [7], and geophysical sciences [8]. IceCube operates continuously and routinely achieves over 99% up-time while simultaneously being sensitive to the whole sky. Among the highlights of IceCube's scientific results is the discovery of an all sky astrophysical neutrino flux and the detection of a neutrino from Blazar TXS 0506+056 that triggered follow-up observations from a slew of other telescopes and observatories [3, 4].

✉ Hendrik Schwanekamp
hendrik.schwanekamp@gmx.net

✉ Ramona Hohl
rhohl@nvidia.com

✉ Benedikt Riedel
briedel@icecube.wisc.edu

¹ NVIDIA Corp., 2788 San Tomas Expressway, Santa Clara, CA 95051, USA

² Wisconsin IceCube Particle Astrophysics Center, University of Wisconsin-Madison, Madison, WI 53706, USA

³ Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, USA

⁴ Department of Computational Mathematics, Science and Engineering, Michigan State University, East Lansing, MI 48824, USA

⁵ Department of Physics, University of Maryland, College Park, MD 20742, USA

⁶ Institut für Physik, Johannes Gutenberg-Universität Mainz, Staudingerweg 7, Mainz 55128, Germany

⁷ DESY, Zeuthen 15738, Germany

Neutrinos that interact with the ice close to or inside of IceCube, as well as cosmic ray interactions in the atmosphere above IceCube, produce secondary particles (muons and hadronic or electromagnetic cascades). Such secondary particles produce Cherenkov light [10] (blue as seen by humans) as they travel through the highly transparent ice. Cherenkov photons detected by DOMs can be used to reconstruct the direction, type, and energy of the parent neutrino or cosmic ray air shower [11].

The propagation of these Cherenkov photons has been a difficult process to model, due to the optical complexity of the ice in the Antarctic glacier. Developments in IceCube's simulation over the last decade include a powerful approach for simulating "direct" propagation of photons [9] using GPUs. This simulation module is up to 200× faster than a CPU-based implementation and, most importantly, is able to account for some of the ice properties such as a shift in the ice properties along the height of the glacier [12], and varying ice properties as a function of polar angle [13, 14], that are crucial for performing precision measurements and would be extremely difficult to implement with a table-based simulation that parametrizes the ice properties in finite volume bins [9].

IceCube Upgrade and Gen2

The IceCube collaboration has embarked on a phased detector upgrade program. The initial phase of the detector upgrade—IceCube Upgrade—is focused on detector calibration, neutrino oscillation studies, and research and development (R&D) of new detection modules with larger light collection area. Planned subsequent phases—IceCube-Gen2—will focus on high-energy neutrinos (>10 TeV), including optical and radio detection of such neutrinos. This will result in an increase in science output, and a corresponding increase in need for Monte Carlo simulation and data processing.

The detector upgrades will require a significant retooling of the data processing and simulation workflow. The IceCube Upgrade will deploy two new types of optical sensors and several potential R&D devices. IceCube-Gen2 will add another module design and require propagating photons through the enlarged detection volume. The photon propagation code will see the biggest changes to its algorithms to support these new modules and will require additional optimizations for the up to 10× larger detector volume.

Photon Propagation

The photon propagation follows the algorithm illustrated in Fig. 1. The simplistic mathematical nature, common algorithm among all photons, the ability to propagate photons independently, and potential to use single-precision floating point mathematics allow for massive parallelization using either a large number of CPU cores or GPUs; see Fig. 2.

The algorithm consists of the following parts. The trajectories of Cherenkov-emitting particles (e.g., tracks with a charge) or calibration light sources are split into *steps*. Each *step* represents a region of close-to-uniform Cherenkov emission properties, i.e., of constant particle velocity v/c within reasonable limits. The *step* length and its velocity are then used to determine the average number of Cherenkov photons to be emitted. This average is used, in turn, to sample from a Poisson distribution to generate an integer number of photons to be emitted from the *step*. All photons from a *step* will be emitted at the same Cherenkov angle. A *step* thus consists of a position, a direction, a length, a particle speed, and an integer number of photons to be created from it, and is used as the basic unit of light emitter in the propagation kernel. The *step* concept can also be used to directly create point emitters using a special mode where a *step* length of zero is specified. The propagation kernel will treat the *step* as a non-Cherenkov emitter in this case and all photons inherit the *step* direction directly. This mode can be used to describe light emission of calibration devices such as "flasher" LEDs. In general, steps are either created by low-level detailed particle propagation tools such as Geant4 or are sampled from distributions when using fast parameterizations.

The number of photons inserted along the path depends on the energy loss pattern of the secondary particles. Most higher energy particles will suffer stochastic energy losses due to bremsstrahlung, electron pair production, or ionization as they travel through the detector, causing concentrations of light at certain points along the particle's path; see red circles in Fig. 3. For calibration sources, a fixed number of photons are inserted depending on the calibration source and its settings.

These *steps* are then added to a queue. For a given device, a thread pool is created depending on the possible number of threads. If using a CPU, this is typically one thread per logical core. When using a GPU, this mapping is more complicated, but can be summarized as one to several threads per "core". The exact mapping depends on the specific vendor and architecture, e.g., NVIDIA CUDA cores are not the same as AMD Stream Processors and the CUDA cores on Ampere architecture cards, e.g., NVIDIA RTX 3080 [15], are different from those on Pascal architecture cards, e.g., NVIDIA GTX 1080 Ti [16]. Each thread takes a *step* of the queue and propagates it. The propagation algorithm first samples the absorption length for each photon, i.e., how long the photon travels before it is being absorbed. Then, the algorithm determines the distance to the first scattering site and the photon is propagated for that distance. After the propagation, a check is performed to test whether the photon has reached its absorption length or intersected with an optical detector along its path. If the photon does not pass these checks, the photon is scattered, i.e., a scattering angle and a new scattering distance are drawn from the corresponding distributions, anisotropy effects are accounted for, and the cycle repeats. Once the photon has either been absorbed or intersected with an optical detector, its propagation is halted and the thread will take a new photon from the *step*.

Fig. 1 Photon propagation algorithm **Fig. 2** Thread execution model for photon propagation [9]

Fig. 3 Cartoon showing the path of photons from a calibration light source embedded in optical sense (left) and a particle (right) to the optical sensors [9]

Fig. 4 Physical aspects of the medium. Left is the effective geometric scattering lengths of light in ice ($b_e = 1/\lambda_e$) as a function of depth for two optical models (SPICE MIE and AHA) [12]. Right is ice anisotropy as seen in the intensity of received light (as measured

The IceCube photon propagation code is distinct from others, e.g., NVIDIA OptiX [17] in that it is purpose-built. It handles the medium, i.e., glacial ice and the physical aspects of photon propagation, e.g., various scattering functions [12], in great detail, see Fig. 4. The photons traverse through a medium with varying optical properties. The ice has been deposited over several hundreds of thousands of years [18]. Earth's climate has changed significantly during that time and imprinted a pattern on the ice as a function of depth. In addition to the depth-dependent optical properties, the glacier has moved across the Antarctic continent [19] and has undergone other unknown stresses. This has caused layers of constant ice properties to be, optically speaking, tilted, and anisotropic.

IceCube's History with CUDA and OpenCL

There are currently two different implementations of the photon propagation code: *Photon Propagation Code* (PPC) and *CLSim*. PPC was the first implementation of the

by the ratio of observed charge in a simulation using an isotropic ice model and data) between simulation and data as a function of orientation [14]

photon propagation algorithm, written initially in C++ and Assembly for CPUs, then in CUDA for GPU acceleration. For Monte Carlo production workloads, the collaboration relies on *CLSim*. *CLSim* was the first OpenCL port of PPC, designed during a time when IceCube's resource pool was a mix of AMD and NVIDIA GPUs. PPC has since also been ported to OpenCL, and is predominately used for R&D of new optical models. This paper will focus on *CLSim* as the propagator for production workloads, where optimizations matter the most.

Over the last several years, NVIDIA GPUs have dominated new purchases, and are now the vast majority of IceCube's resource pool. This makes CUDA a valid choice again for production workloads. However, with AMD and Intel GPUs both being chosen for the first two US-based Exascale HPC systems, both the CUDA and OpenCL implementations will have their use cases in the future.

Optimizing Performance with CUDA

In an attempt to speed up the simulation, the OpenCL part of the code was ported to CUDA. While this only works on NVIDIA GPUs, we gain access to a wide range of tools for profiling, debugging, and code analysis. It also enables us to use the most recent features of the hardware. As described in Sect. 2, the algorithm is quite complex. It consists of multiple parts (photon creation, ice propagation, collision detection, and scattering) that all utilize the GPU in different ways and can hardly be executed on their own.

The data parallel compute model of GPUs requires a different computing model than the traditional CPUs. GPU threads are not scheduled for execution independently, as done with “traditional” threads on CPUs. GPU threads are dispatched to the hardware in groups of 32 threads, the so-called *warps* in CUDA. For a warp to be completed, all 32 CUDA threads need to have finished their computation. At the same time, threads within a warp can take different code paths, e.g., at an *if-then-else* statement or loop. This creates the potential for one or more CUDA threads in a warp to affect the execution characteristics of a warp, i.e., warp wastes computing time needing to wait for all the 32 CUDA threads to complete. This effect is called *warp divergence* [20]. While some divergence is often unavoidable, having much of it will hurt performance. In the photon propagation case, photons in different CUDA threads within a warp take different paths through the ice; hence, their execution time can vary significantly within a warp. Thus, creating said warp divergence.

We focused on changes to the GPU kernel itself, and assumed that the CPU infrastructure provides enough work to keep the GPU busy at all times. For the optimization of such complex kernels, we developed a five-phase process. It is general enough to also be applied to other programs and hardware.

Initial Setup

To monitor progress during the optimization process, it is important to ensure that code changes do not introduce changes to the output and to quantify the performance gains or losses. Due to the stochastic nature of the simulation, results cannot be directly compared. We employed a Kolmogorov–Smirnov test as a regression test to verify changes to the code, by comparing the distribution of detected photons across implementations; see Fig. 5 for an example.

To measure performance, two different benchmarks were used. Photons are bundled in steps. Every GPU thread then handles photons from one step. Therefore, one benchmark contains steps of equal size and the other introduces more imbalance.

In addition, we focused on a single code path. Removing compile-time options for other code paths helps to focus on the important parts and not waste effort.

First Analysis and Optimization

This phase of the optimization focuses on clearly identified problems and changes that can be implemented quickly. Depending on the kernel in question, they might include the following:

- Launch configuration and occupancy.
- Math precision (floating point number size, fast math intrinsics).
- Memory access pattern and use of correct memory type (global, shared, constant, local).
- Warp divergence created by control flow constructs.

A profile taken with Nsight Compute gave an overview of the current state of the code. It showed only medium occupancy and problems with the kernel’s memory usage. We tackled the occupancy by finding a better size for the warps and removing unused debugging code, which reduces register usage. It also made the code easier to read and analyze in the following steps.

To improve the way the kernel uses the memory system, we used global memory instead of constant memory for look-up tables. Those are usually accessed randomly at different addresses by different threads of a warp. The constant memory is better suited when threads of a warp access the same, or only a few different memory locations [21].

In-Depth Analysis

In a complex kernel, it can be difficult to identify the cause of a performance problem. This phase focuses on collecting as much information as possible about the kernel to make informed decisions later during the optimization process.

The most important tool to use here is a profiler that provides detailed information on how the code uses the available resources. To see where most of the time is spent in a large kernel, parts of the code can be temporarily disabled. CUDA also provides the `clock64()` function, which can be used to make approximate measurements of how many clock cycles

Fig. 5 Plot showing two baseline comparison quantities, total number of photons observed and photon arrival time at a detector, used for the Kolmogorov–Smirnov test

a given section of code takes. Both methods produce rough estimates at best, but can still be useful when deciding on which part of the kernel to consider for optimization.

In addition, custom metrics can be collected. For example, the number of iterations performed in a specific loop can be measured, or how many threads taken in a particular code path. This helps to identify where the majority of warp divergence is caused, and which parts of the code are more heavily used.

From the analysis of the IceCube kernel, we learned that propagation and collision testing each make up for one-third of the total runtime. When a photon is created, it will be scattered, propagated, and tested for collision 26 times on average. The chance of a collision occurring in a given collision check is 0.001%. This explains why photon creation and the storing of detected collisions (hits) have a smaller impact on overall runtime.

Warp divergence was also a major problem. Only 13.73 out of 32 threads in each warp were not masked out on average. A test showed that eliminating warp divergence would speed up the code by a factor of 3. In this test, all threads calculate the same photon path, which does not produce useful simulation results. Eliminating all divergence in a way that does not effect simulation results is not realistic, as some divergence is inherent to the problem of photon propagation (different photons need to take different paths). The test merely demonstrates the impact of warp divergence.

Major Code Changes

Guided by the results from the previous analysis, in this phase, time-consuming code changes can be explored. For a particular performance problem, we first estimate the impact of an optimization by fixing the problem in the simplest (potentially code breaking) way possible and measure. That ensures that the source of the problem was identified correctly; results show whether or not a proper implementation will be worth it to fully implement.

If possible, the function in question is then implemented using a high-level language (e.g., Python), in a single-threaded context. This allows for easier testing and debugging of the new optimized version of the function. Once it gives the same results as the original implementation, it is ported back to CUDA. Then, it is profiled again and the regression test is performed to make sure that the code is correct and the changes result in the expected performance improvement.

To combat the high warp divergence, we implemented a warp level load-balancing algorithm, which we call *Per Warp Job-Queue*. A list of 32 GPU thread inputs (photons in the case of IceCube) are stored in shared memory. Each thread performs one iteration of photon propagation. If its photon is absorbed or hits a sensor, the thread loads the next photon from the list. When the list is empty, all 32 threads come together to load new photons from the step queue and store them in shared memory.

The algorithm is explained in detail in Appendix 6. The *Per Warp Job-Queue* yields a speed up when the the number of photons varies between steps, i.e., the workload is unbalanced. With an already balanced workload, i.e., all steps contain an equal number of photons, the algorithm does not yield a meaningful benefit. Rather, we add the overhead, e.g., memory look-up, for the list, and the algorithm to sort the workload *a priori*. During this phase, we also experimented with NVIDIA OptiX to accelerate the collision detection, as detailed in Sect. 4.

Incremental Rewrite

When an even more detailed understanding of the code is required to make new optimizations, the incremental rewrite technique can be employed. Starting from an empty kernel, the original functionality is restored step-by-step. Whenever the runtime increases considerably, or a particular metric in the profile seems problematic, the new changes are considered for optimization. This forces the programmer to take a second look at every detail of the kernel.

One problem with this technique is that regression tests can only be performed after the rewrite is complete. At this point, many modifications have been made and bugs might be hard to isolate. The process is also quite time-consuming.

In the IceCube code, the photon creation as well as the propagation part require data from a look-up table. Bins in the table have different widths and linear search is used to find the correct bin every time the table is accessed. The data are also interpolated with the neighboring entries in the table. We re-sampled both look-up tables as a pre-processing step on the CPU, using bins of regular size and higher resolution. This eliminated the need for searching and interpolation, reducing the number of computations, memory bandwidth usage, and warp divergence.

Warp divergence was reduced further by eliminating a particularly expensive branch in the propagation algorithm. The algorithm to determine the trajectory of an upgoing (traveling in $z > 0$ in terms of detector coordinates) or downgoing ($z < 0$) photon is the same *modulo* a sign. To reduce the branching behavior, we introduced a variable “direction”, that is set to 1 or -1 depending on the photon’s movement direction in z . It is then multiplied with direction-dependent quantities during the propagation rather than following a different code path.

Divergence is also introduced when branches or loops depend on random numbers, as threads in the same warp will usually roll differently. This is particularly expensive for the distance to the next scattering point. It uses the negative logarithm of the random number, which results in occasional very large values. To alleviate this divergence, we generated a single random number in the first thread of each warp, and use the `shuffle()` function from the cooperative groups API [22] to communicate it to the other 31 threads. This optimization needs to be explicitly enabled at compile time, as it might impact the physical results.

Finally, we used the remaining free shared memory to store as many look-up tables as will fit. While reads from global memory are cached in L1 (which uses the same hardware as shared memory), data could be overwritten by other memory accesses before the look-up table is used again. In contrast, data in shared memory are persistent during the execution of a thread block. Unfortunately, that means the same data are copied to and stored in shared memory multiple times, once for each thread block. It is, however, the closest we can get to program-managed L1 cache with the current CUDA version.

OptiX

The NVIDIA OptiX [17] ray tracing engine allows defining custom ray tracing pipelines and the usage of RT cores. RT cores accelerate bounding volume hierarchy traversal and triangle intersection. In OptiX, ray interactions are defined in *programs*. These include the *closest hit program*, defining the interaction of a ray with the closest triangle, and the *intersection program*, which enables to define intersection with custom geometry. Rays for tracing the geometry acceleration structure are launched within the *ray generation program*. The *programs* are written in CUDA and handled by OptiX in form of PTX code, which is further transformed and combined into a monolithic kernel. The OptiX runtime handles diverging rays (threads) by fine-grained scheduling. For a detailed description of the OptiX execution model, see [17, 23].

Besides any benefit in current performance, an OptiX-based implementation is also important as a prototyping tool. For the ongoing detector upgrades, see Sect. 1.1, OptiX allows us to add arbitrary shapes, such as the pill-like shape of the D-Egg, to the Monte Carlo workflow without any significant code changes. This will be important as the R&D for IceCube-Gen2 continues to ramp up and multiple new photo-detector designs will be tested.

Collision Detection with OptiX

For each propagated photon path, it is checked for intersection with an optical detector. If that is the case, a hit is recorded and written to global memory; see Fig. 1. In the OptiX version, rays are traced defined by their propagated path, and hits are recorded in the *closest hit program*. When including the cables (which absorb photons) in the simulation, we mark the ray as absorbed in the *closest hit program* when it hits part of the cable geometry. In the CUDA and original OpenCL implementation, the cables are accounted for through decreasing the acceptance of individual detectors by some factor, i.e., the cables cast a “shadow” onto the individual detectors that reduces their light collection area. This averages out the effect of the cables across the detector, but for individual events

or certain energy ranges, in particular neutrino oscillation analysis using atmospheric neutrinos, a more precise implementation, e.g., position of cable with respect to sensor and neutrino interaction, would aid in accounting for detector systematic effects.

The performance results of this OptiX version run on an *RTX6000* are 0.55× the speed of the CUDA code. Nevertheless, we emphasize the flexibility this implementation offers. We have tested different geometry models; Fig. 6 shows an illustration of the two different detectors and the pill-shaped model with cables. For example, one can now observe the effect of the cable on the number of detected hits by simply providing the according cable geometry in the form of a triangulated mesh. Each detector is also represented by triangles, and increasing the resolution of the mesh gives more accurate results. Exact hit points can be attained by custom sphere intersection, in case of spherical detectors. Table 1 lists the mesh complexity, ray hits, and runtime for the different detector geometries, for a total of 146,152,721 rays run on an *RTX6000*. The runtime is stable for different types of geometries despite different numbers of ray hits.

This version of the photon propagation simulation allows for simple definition of detector geometries without rewriting or recompiling any code. It can be used to simulate future detectors in a test environment before implementing changes in production code.

Results and Conclusion

This paper shows how the IceCube photon propagation code was ported to CUDA and optimized for use with NVIDIA GPUs. Section 3 explains most changes we made to the code. The results can be seen in Fig. 7. It compares the runtime of different versions of the code on various GPUs. The CUDA version running on a recently released *A100* GPU is 9.23 × faster than the *OpenCL* version on a *GTX1080*, the most prevalent GPU used by IceCube today. A speedup of 3 × is achieved by the optimized CUDA version over the *OpenCL* version on an *RTX6000* and 2.5 × on an *A100*.

The code is mostly bound by the computational capabilities of the GPU and only uses 29.05% of the available memory bandwidth. The amount of parallel work possible is directly proportional to available memory on the GPU, i.e., enough photons to occupy the GPU will be buffered in memory, and to the amount of *steps* (groups of photons with similar properties) simulated, i.e., the higher the energy of the simulated events, the higher the computational efficiency. Therefore, we are confident that the simulation will scale well to future generation GPUs. Not all of the changes made in CUDA can be backported to the original *OpenCL* code. The biggest hurdle for this is the *Per Warp Job-Queue* algorithm that depends on CUDA-specific features.

Fig. 6 Spherical- and pill-shaped detector models

The version of the code that uses NVIDIA OptiX is slower than the CUDA code for the current detector geometry and physics model. It is still useful, however, as it makes it simple to experiment with different geometries, detector layouts, or detector modules, as will be deployed in the IceCube-Upgrade and planned for IceCube-Gen2 (see Sect. 1.1).

There are several potential pathways for future improvements. One of them aims to further improve the performance of the CUDA code, in particular the collision detection part. It might be possible to use the CUDA code for propagation and then send the photon paths to OptiX to test for collision.

While explaining the changes made to the IceCube code, we also generalized our optimization process, so it can be applied to other complex GPU kernels. We found that it is important to perform detailed measurements and test often, as it allows for making the best decisions on where to change the code. To use the development time as efficiently as possible, one should focus on quick to implement optimizations in the beginning and slowly progress to more complicated ones. Our five-phase process can be used as a guide when working on the performance optimization of complex GPU kernels.

A Per Warp Job-Queue

The *Per Warp Job-Queue* can be used whenever GPU work items with different length are grouped into packages of different size. To reduce imbalance between the threads, each warp collaborates on one package. Individual items are

Fig. 7 Runtime comparison

processed in a loop, requiring a different amount of iterations each. The input could therefore be processed without additional divergence between the GPU threads. Once a thread finishes work on its current item, it retrieves a new one from a list in shared memory. An atomically accessed integer in shared memory is used to keep track of the number of items left. When the list is empty, the threads fill it with new items from the package. Listing 1 shows the pseudo-code of the algorithm.

Table 1 Hits and kernel runtimes for different detector geometries

Model	# Triangles	DOM hits	Cable hits	Total hits	ns/ray
Sphere DOMs	9'907'200	42'099	0	42'099	3.94
Sphere DOMs, cables	9'926'640	40'978	132'209	173'187	3.94
Simple pill DOMs	12'291'120	61'437	0	61'437	3.89
Simple pill DOMs, cables	12'310'560	60'250	128'797	189'047	3.94

Algorithm 1: Per Warp Job-Queue

```

load item from bundle;
fill queue with items from bundle;
while thread has valid item do
  do work on current item;
  if not queue empty or not bundle empty then
    if not thread has valid item then
      | get item from queue;
    end
    if queue empty then
      | fill queue with items from bundle;
      if not thread has valid item then
        | get item from queue;
      end
    end
  end
end
end

```

If the processing of an input is not iterative, and all computations take approximately the same time, the pattern can be simplified accordingly. The same applies if creating a input from the set of inputs in place is faster than loading it from shared memory. In this case, the queue stored in shared memory would not be needed.

As the thread group holds 32 elements at a time, one would expect that rounding the number of inputs in each set to a multiple of 32 might improve the performance. However, that is not necessarily the case in practice. It might make a difference in a program, where loading or generating a work item from the bundle is more time-consuming, or when the amount of items per bundle is low. What does improve the performance of the code in case of IceCube, however, is using bundles that contain more work items.

Data Availability Statement The datasets generated and analysed during the current study are available from the corresponding author on reasonable request.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated

otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Aartsen MG et al (2017) The IceCube Neutrino Observatory: Instrumentation and Online Systems. JINST 12(03):P03012. <https://doi.org/10.1088/1748-0221/12/03/P03012> (1612.05093)
2. Aartsen MG et al (2013) Evidence for High-Energy Extraterrestrial Neutrinos at the IceCube Detector. Science 342:1242856. <https://doi.org/10.1126/science.1242856> (1311.5238)
3. Aartsen M et al (2018) Neutrino emission from the direction of the blazar TXS 0506+056 prior to the IceCube-170922A alert. Science 361(6398):147–151. <https://doi.org/10.1126/science.aat2890>
4. Aartsen M et al (2018) Multimessenger observations of a flaring blazar coincident with high-energy neutrino IceCube-170922A. Science 361(6398):eaat1378. <https://doi.org/10.1126/science.aat1378>
5. Aartsen MG et al (2015) Search for dark matter annihilation in the galactic center with IceCube-79. Eur Phys J C75(10):492. <https://doi.org/10.1140/epjc/s10052-015-3713-1> (1505.07259)
6. Aartsen MG et al (2020) Cosmic ray spectrum from 250 TeV to 10 PeV using IceTop. Phys Rev D 102:122001. <https://doi.org/10.1103/PhysRevD.102.122001> (2006.05215)
7. Aartsen MG et al (2016) Searches for sterile neutrinos with the IceCube detector. Phys Rev Lett 117(7):071801. <https://doi.org/10.1103/PhysRevLett.117.071801> (1605.01990)
8. Collaboration I (2013) South pole glacial climate reconstruction from multi-borehole laser particulate stratigraphy. J Glaciol 59(218):1117–1128. <https://doi.org/10.3189/2013JG13J068>

9. Chirkin D (2013) Photon tracking with GPUs in IceCube. *Nucl Instrum Meth A* 725:141–143. <https://doi.org/10.1016/j.nima.2012.11.170>
10. Cherenkov P (1936) *Cr acad. sci. ussr*, 8 (1934) 451. In: *Dokl. AN SSSR*, vol 3, p 14
11. Aartsen M et al (2014) Energy Reconstruction Methods in the IceCube Neutrino Telescope. *JINST* 9:P03009. <https://doi.org/10.1088/1748-0221/9/03/P03009> (**1311.4767**)
12. Aartsen MG et al (2013) Measurement of South Pole ice transparency with the IceCube LED calibration system. *Nucl Instrum Meth A* 711:73–89. <https://doi.org/10.1016/j.nima.2013.01.054> (**1301.5361**)
13. Chirkin D (2013) Evidence of optical anisotropy of the South Pole ice. In: 33rd international cosmic ray conference, p 0580
14. Chirkin D, Rongen M (2020) Light diffusion in birefringent polycrystals and the IceCube ice anisotropy. *PoS ICRC2019*:854. 10.22323/1.358.0854 **1908.07608**
15. NVIDIA (2019) Nvidia 3080. <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080/>
16. NVIDIA (2019) Nvidia 1080 ti. <https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/> Accessed 26 Feb 2021
17. Parker SG, Bigler J, Dietrich A, Friedrich H, Hoberock J, Luebke D, McAllister D, McGuire M, Morley K, Robison A, Stich M (2010) Optix: a general purpose ray tracing engine. *ACM Trans Graph.* <https://doi.org/10.1145/1778765.1778803>
18. Price PB, Woschnagg K, Chirkin D (2000) Age vs depth of glacial ice at south pole. *Geophys Res Lett* 27(14):2129–2132. <https://doi.org/10.1029/2000GL011351>
19. Rignot E, Mouginot J, Scheuchl B (2011) Ice flow of the Antarctic ice sheet. *Science* 333(6048):1427–1430. <https://doi.org/10.1126/science.1208336>
20. Corporation N (2021) CUDA C++ Programming Guide, 11.2. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed 26 Feb 2021
21. Corporation N (2021) CUDA C++ Best Practice Guide, 11.2. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>. Accessed 26 Feb 2021
22. Harris M, Perelygin K (2017) Cooperative Groups: Flexible CUDA Thread Programming. <https://developer.nvidia.com/blog/cooperative-groups/>
23. Parker SG, Friedrich H, Luebke D, Morley K, Bigler J, Hoberock J, McAllister D, Robison A, Dietrich A, Humphreys G, McGuire M, Stich M (2013) Gpu ray tracing. *Commun ACM* 56(5):93–101

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.