Automatically parallelizing Diderot programs on CUDA targets

Adrian E. Lehmann

March 3, 2023

Contents

List of Figures

List of Tables

Abstract

Diderot is a domain-specific language to perform scientific visualizations. Its programs are structured largely like bulk-synchronous parallelism. In this pattern, multiple strands (often also called treads) run one update step in isolation, followed by a single global reduction step (similar to MapReduce). Currently, a compiler exists that transforms Diderot programs, along with the domain-specific operations, into C++. The compiler supports targeting both sequential and parallel CPU execution models. However, given the programming model's parallel nature, adding GPU support to Diderot's compiler is a natural step.

Our work fills this gap. We add support for automatically parallelizing Diderot applications by modifying the compiler to be able to generate CUDA code. We propose three strategies to schedule CUDA threads. One that closely follows the BSP model, one that runs strands to completion (assuming no reduction steps are needed), and one that builds on a work queue. We also propose a permutation mechanism for stochastic load distribution to mitigate strand divergence. We also create variants that utilize CUDA unified memory, an API to move memory pages between system and GPU memory.

In benchmarks, we see speedups of 60-500x, where the queue-based approach outperforms other approaches. Further, we see differences in the performance of our approaches between benchmarks. We observe that permutation performance is highly dependent on the benchmark structure and the homogeneity of strand execution. Furthermore, we conclude that in our test, CUDA unified memory leads to a significant performance penalty for benchmarks with fewer strands while greatly simplifying the produced code.

Chapter 1

Introduction

In this paper, we look at automatic CUDA code generation for Diderot, a domain-specific language for scientific computation. Diderot has a lot of unique features useful for its domain: higher-level operators, continuous field representation, tensors, and more. Features like this make Diderot very expressive and useful for the domains it targets Diderot's computation model is based on the bulk synchronous parallelism (BSP) model: The program is divided up into strands, each of which has an update step. Each strand's update step can be executed concurrently and solely depends on global read-only data. Further, there exists a global reduction step, which is executed after all strands complete their update steps. In this step, global variables may be modified. This model captures the abstraction of GPU parallelism for common scientific tasks well.

Writing out code to perform these operations manually is cumbersome and requires a lot of specialized knowledge. By providing a higher-level language to handle the implementation details, it frees the domain expert to focus on the development of their actual system. Furthermore, adding on the layers of parallelism (CPU or GPU) increases complexity tremendously. It often takes highly knowledgeable people in the area of parallel computing to efficiently implement a parallel version of the task at hand. Diderot's goal is, however, to let the domain experts write their own software, without having to worry about implementation details. One could argue, that any compiler engineer fundamentally wants to hide complexity, to allow the programmer to focus on the task at hand while having the compiler take care of the underlying details. Of course, if it were easy to completely hide complexity, we would not need experts to write GPU code. This is where the restricted programming model of a DSL, such as Diderot, comes in. It allows us to tailor our approach to compilation to a specific set of problems and exploit the structure of such programs to produce more efficient results.

In its current state, Diderot has a compiler that lowers all of the higher-level operations and the BSP structure to C++ code. The C++ code can be either sequential or CPU parallel. Previously there was also a GPU target for OPENCL. This target, however, was ultimately discontinued due to poor performance and lack of features. Our work comes in at this point to add automatic GPU code generation using CUDA. We see the parallelism that Diderot's computation model provides as a great chance to utilize the massively parallel architecture of GPUs. Therefore, in this paper, we will explore different strategies of parallelizing Diderot programs automatically for CUDA hardware. We also take this opportunity to make our work a case study on a CUDA API feature aimed at simplifying memory management between CPU and GPU, called CUDA *unified memory*.

Chapter 2

Prior Work

2.1 Bulk Synchronous Parallelism

When building parallel applications there are different patterns to effectively structure applications to allow for parallelization. One such pattern is called *Bulk Synchronous Paralellism* (BSP) [24]. Programs using BSP are generally structured to have two main components. A bulk function is executed by threads in parallel (we call this the super-step) and a reduction step is executed synchronously. In the most basic form, after initial setup, the stepping and reduction steps execute alternatingly. A commonly known instance of the BSP pattern is MapReduce [9]. Using a BSP pattern allows writing straightforward code that can easily be run in parallel. Experiments have shown that in practice BSP gives great performance as it is akin to how parallel hardware works. This makes it a great pattern for parallel computation as complicated synchronization structures are very difficult to optimize. Further, it allows for many high-level language features to be used.

2.2 GPU compute & CUDA

Using graphics cards for general-purpose parallel computation has been a pattern for a while. On NVIDIA GPUs the API for such computation is called *Compute Unified Device Architecture* (CUDA) [**?**]. CUDA is designed to work with C/C++ by providing an API allowing one to execute functions, called *kernels* in parallel. These kernels are executed across CUDA *cores* (which can be thought of as mini CPUs), each executing a function while having access to the shared VRAM. CUDA structures this execution by arranging threads in a two-layer nested grid. Specifically, CUDA has a 2D grid of *blocks*, where blocks contain a 2D grid of threads. We show the structure of a nested grid in Figure 2.1. A block is always executed by one *streaming multiprocessor* (SM) on GPU, which is a processor that has multiple individual CUDA cores. Each CUDA capable GPU supports arbitrarily sized blocks and grids up to a fixed maximum block and grid size [11]. When executing a kernel users can specify block and thread per block count. Here a user can arrange their grid in 1D, 2D, or 3D space, which CUDA translates down to a 2D grid to align with the hardware.

CUDA SMs each have their own read-only memory for the instruction and texture cache, registers, and L1 cache. All SMs share a common L2 Cache and the VRAM. This means that aligning tasks with similar data on a single SM is advantageous, however, one can still have cache effects computing across multiple SMs.

32 consecutive threads on a given SM are called a *warp*. Warps are the basic unit of execution on an SM. In an SM there is then a hardware scheduler that schedules warps to be active. Modern CUDA GPUs also have advanced features in warp scheduling, such as overprovisioning to reduce instruction and data latency [18]. Even with these new systems in place, the concurrent execution of 32 threads at a time points to an issue that an application developer needs to address: *Warp divergence*. Warp divergence occurs when some threads in a warp have completed work while others are still running. Have warp divergence, with many warps still having some threads with work can lead to great performance penalties.

Besides computation CUDA also provides control over VRAM memory management. In general, CUDA treats GPU VRAM memory as distinct from system memory or other storage in a computer. This means that CUDA provides a dedicated address space for the GPU memory and has no swapping mechanisms. In this model, developers can explicitly copy chunks of memory synchronously or asynchronously to GPUs. Note that CUDA allows arbitrary amounts of data

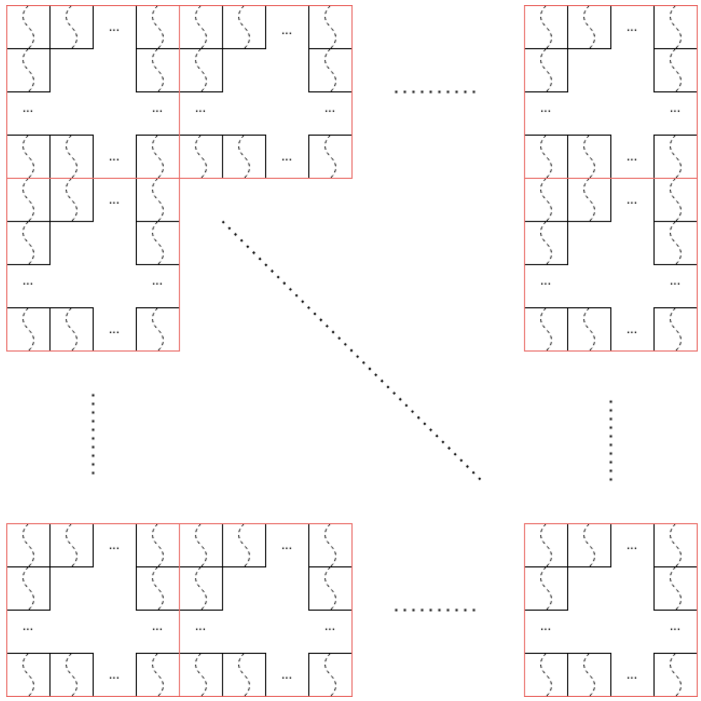


Figure 2.1: The layout grid of CUDA threads. A square with a dotted squiggly line represents a thread, a red square represents a block.

(as long as they fit into VRAM constraints) to be copied, even though PCIe connections used to connect the GPU to the CPU only allow fixed-size chunks (256MB) of memory to be copied[1].

There are a few limitations of this approach: The GPU only has limited memory, usually a fraction of the system memory. This means that developers often need to implement ways to segment tasks to only require memory within the device's constraints. Furthermore, dynamically interacting with memory on CPU and GPU requires a lot of manual copying and often complex logic to limit copying to small subsets of data to improve execution time. To address these issues NVIDIA introduced CUDA unified memory [12]. Allocating a block of unified memory creates an address space that is accessible from GPU and CPU. Internally this works by dynamically copying pages of this address space between RAM and VRAM (or directly into GPU cache) when required. Creating such a system of course comes with various implications to the application. Even if there are no longer explicit memory copy operations required, memory still has to be copied and the programmer needs to be aware of such patterns. CUDA also provides then option to prefetch memory explicitly to GPU or CPU or hint at access patterns. Moreover, there is some nuance as to how simulataneous memory access from CPU and GPU works. While on older GPUs, simultaneous access is illegal, newer GPUs support CPU access of memory while a kernel is running on GPU. Of course, this comes with a large performance penalty due to the page faults caused and requires explicit synchronization in the application to produce correct results [12].

With all these tools a programmer can then write efficient GPU programs with low level control of the hardware, similar to the capabilities the host language C++ offers for CPU. To write such efficient programs, however, the programmer has to keep in mind that due to the vast architecture differences, there are a few key considerations when programming for the GPU.

1. GPUs have an extremely parallel architecture, meaning that like for CPU multi-threaded programming any avoided synchronization results in performance gains. Given the low performance of individual execution units on GPUs compared to CPU cores but much higher number of said cores programs need to often be fundamentally rearchitected.

2. As discussed before, memory access and moving of data between CPU and GPU is a large new layer of performance consideration, as transfer operations between CPU and GPU both have high latency and comparatively

[1] Recently *resizable Base Address Register (BAR)* is a PCIe standard introduced to allow arbitrary size chunks to be copied

low bandwidth.

3. Similarly, GPUs usually have much less memory than the host system does. This creates additional limitations and often requires different memory layouts. Furthermore, such a massively parallel architecture memory access patterns can lead to bottlenecks of memory access due to the high bandwidth requirement. Hence in practice, it is often advantageous to only let kernels access the memory

4. Most data structures typically used on CPU do not have sufficient synchronization operations and most even if these are added cause drastic performance bottlenecks due to frequent and global synchronization requirements. Hence, different data structures have to be used for GPU programs.

2.3 Diderot

Our work builds on Diderot [16], a domain-specific language (DSL) for scientific analysis and visualization. For this, it natively supports a variety of operations targeting scientific analysis. Most of these operations include matrix, vector, and tensor field operations, with support for higher-order operations. Diderot's goal is to allow developers to write concise, expressive code that can be then either used on its own or integrated into a larger project as a library. A Diderot program is then compiled to C++ to either produce a standalone executable or a library that can be linked into any language supporting C-like libraries.

More relevant to our work Diderot uses a very interesting parallel model, based on the BSP model. Diderot programs contain a super-step and a reduction step. Diderot calls the super-step `update` and the reduction step `global`. In the `global` step all strands' values can be used to update global variables that are available to all strands in the next super-step or the program can be terminated. Diderot separates its execution into threads, called `strands`. During the super-step, all strands have their update functions called to update their strand's state. For this calculation, they can access the results of strands from the last super-step. A key addition to Diderot's parallelism model is that each strand can also have an `initialization` method that is called before the first super-step.

Diderot also allows for the creation and premature deletion of strands. During the execution of a given strand, it can create a new strand making it active from the next execution onward. A strand can also choose to stabilize during its execution, making it idle for the remainder of the program's execution but still keeping its computation result. An example of a Diderot program that finds prime numbers is shown in Figure 2.2.

To implement the aforementioned features, Diderot's runtime system supports three models: Default, indirect and dual. Default is where there are solely global variables and other strand-to-strand communication does not exist. In this case, a global variable section and strand-local section are held in program memory. Hereby each strand gets an entry with its data in a fixed-sized array. In the case of indirect storage, Diderot allows dynamically creating and killing threads. Here it works as above, except that the strand data is held in an unbounded (indirect) array. Lastly, in the case of the dual state, strands can read the last step's state of other strands. This works by having two sets of strand data, one of the previous step and one that can be overwritten for the current step.

Another specialization of the BSP model is how Diderot lays out strands. There are two main patterns, spatial proximity and heap allocation. In Diderot strands can have a position in world space and then lays out the strand's outputs in a data structure that allows for easy sharing of data with strands that are located close to the querying strand. This design choice stems from the fact that in graphics and scientific computation requiring only spatially close data is a common pattern. In practice, strands are allocated in a heap-like structure that aims to have logically close strands close in memory for easier access and data-sharing.

The Diderot compiler takes a Diderot program, optimizes it, and compiles it to C++. Currently, the compiler supports building sequential and parallel versions for the CPU. In the past, it also supported compiling to GPU by compiling to OpenCL. Though, the OpenCL implementation was not performant and did not receive updates for more advanced features of Diderot. Hence, it was discontinued and with this work, we help to fill in the gap left by this deprecation with a performant CUDA implementation.

The compiler is divided into a few phases. First, it takes in the Diderot program and strips syntactic sugar and compiles high-order functions down. Then it uses the result to optimize mathematical operations. Once the mathematical operations are optimized the compiler breaks down the operations such that they can be converted into C++ code. With the help of predefined headers for common functions the compiler then builds a C++ performing the actions specified by the Diderot program. Depending on the parallel model chosen it also generates appropriate data structures to support execution [5].

```
#version 2.0

input int NN ("highest␣number␣to␣test␣for␣primality") = 100;

int nextp = 2; // first prime to find

// Each strand tests one integer, ii, for primality
strand test(int ii) {
   output int pp = ii;
   update {
      if (nextp == pp) {
         stabilize; // This adds the value nextp to the saved output
      } else if (ii % nextp == 0) {
         die; // Can't be a prime; discard the value
      }
   }
}

update {
   nextp = min { T.pp | T in test.active };
}

create_collection { test(ii) | ii in 2..NN }
```

Figure 2.2: Sieve of Eratosthenes in Diderot

Currently, in the CPU parallel model Diderot works by dividing up stands into blocks of a few strands at a time and then running a step on a block of strands on a given thread. Then a scheduler assigns these blocks of work to any currently idling thread. Threads synchronize after a step completes. We explore an analogous idea in section Section 3.3 for our GPU implementation.

One special feature of the Diderot compiler is that it optimizes the program to target the image used by the program. Hence, this image needs to be provided at compile time. Using the additional information the compiler can them optimize the program to run on the image.

Note that an image does not necessarily have to be a picture in this context. With image data, virtually any continuous field can be modeled.

2.4 Other languages for GPU scientific analysis and visualization

Shadie Shadie is a domain-specific language targeted at volumetric rendering. Hence, unlike Diderot, Shadie divides up its units of parallelism into rays and essentially performs specialized raycasting. Like Diderot, however, Shadie is able to handle data in the form of continuous fields. This means it is a less general language than Diderot, as Diderot's applications go beyond volumetric rendering. Shadie, however, has GPU support already, specifically it has a compiler that generates CUDA code [**?**].

Vivaldi Vivaldi is a DSL, like Shadie, for volume rendering. In contrast to Shadie and Diderot, Vivaldi its language is less specialized to the domain and rather a python-esque language with special functions for volume rendering. It does not support higher-level operations or continuous fields. Vivaldi's compiler also supports GPU targets, specifically CUDA and OPENCL. In stark difference to the approach we outline in Chapter 3 and Shadie's approach to GPU code generation, Vivaldi outputs python code with api calls to APIs wrapping GPU execution [8].

Scout Scout is a DSL that aims to speed up data-parallel programs by compiling to GPU. Unlike Diderot, it focuses on a voxel-based approach, where multiple voxels are grouped in definable ways to create parallelism. It also does not support higher-order operators, though it does offer a limited set of complicated mathematical operations as library functions. Given that scout was built in 2007, the pipeline to create GPU (CUDA) code is quite different and more low-level than more recent work. It does show, however, that GPU compute applied to scientific visualization tasks can yield significant speedups [20].

Overview In the previous section we review other work that aims to provide GPU execution for scientific visualization and analysis, and contrast it with Diderot. We see that the domain of GPU parallel languages in this domain has been explored before. However, we notice that the other languages do not match the features and flexibility of Diderot. Given the highly different nature of the domain of Scout, Shadie, and Vivaldi, we do not think it is possible to draw fair comparisons between their GPU implementation and the GPU implementation proposed in this paper. We believe that given the overall promising results of these previous papers, extending Diderot with a CUDA code generator is a worthy goal.

Chapter 3

Implementation

3.1 General implementation

The goal is to run Diderot BSP programs on GPU using CUDA. In this paper, we focus on the model of Diderot where there is no strand-to-strand communication, except for global variables and no spawning of new strands. Instead, we choose to focus on the bulk synchronous parallelism and translate it to CUDA. Henceforth we denote the jth strand's ith update invocation as S_j^i and the ith reduction step R^i. Generally, we consider $j \in [1, n]$ and $i \in [1, m]$.

As we discuss in Section 2.2, CUDA arranges threads in a nested 2D grid. Henceforth, $w \times h$ shall be the grid size and $\hat{w} \times \hat{h}$ the block size. When looking at Diderot's model of having n strands, we see that this model translates well into the 2D nested grid model. The question becomes how to map this flat list of strands into a 2D nested grid. Figure 3.3a shows an example of how times mapping happens. In our initial approach, we group $\hat{w}\hat{h}$ strands together into a group of strands denoted by $\hat{S}_j$, where j refers to the index of the group of strands. We denote the number of such group as $\hat{n}$. Note that we leave w and h as parameters and experiment with choosing said parameters. We use these parameters to calculate $\hat{w}$ and $\hat{h}$ to be maximal such that as many of the wh threads as possible have strands assigned.

Now that we have seen how to map the actual strands to the CUDA thread layout we map the bulk synchronous parallelism to GPU computation. As we build our extension in the framework of the Diderot compiler, we recall that with the existing Diderot compiler we can compile sequential code and parallel code. Diderot code generation compiles Diderot programs to C++, complete with all vector space operations, data structures, and general program layout lowered from the higher abstractions of Diderot. We use this code generation, in the sequential variant, as the starting point for building a pipeline to produce CUDA code. Our initial approach is to parallelize each strand's update, then synchronize and perform global reduction, if necessary. We illustrate the computation model in Figure 3.1 and show pseudo-code in Algorithm 1.

Algorithm 1 Standard CUDA implementation

```
function STANDARDCUDAIMPLEMENTATION(nStrands, nCudaThreads)
    blockSize ← nStrands / nCudaThreads
    while There exist alive strands do
        for all i ∈ [0...nCudaThreads) do
            RunThread(i, blockSize)        ▷ In practice this is a cuda call where threads are layed out as described.
        end for
        Barrier synchroize
        Global step
    end while
end function
function RUNTHREAD(threadId, blockSize)
    for all i ∈ [threadId * blockSize, ..., threadId * (blockSize + 1)) do
        Update i^th strand
    end for
end function
```

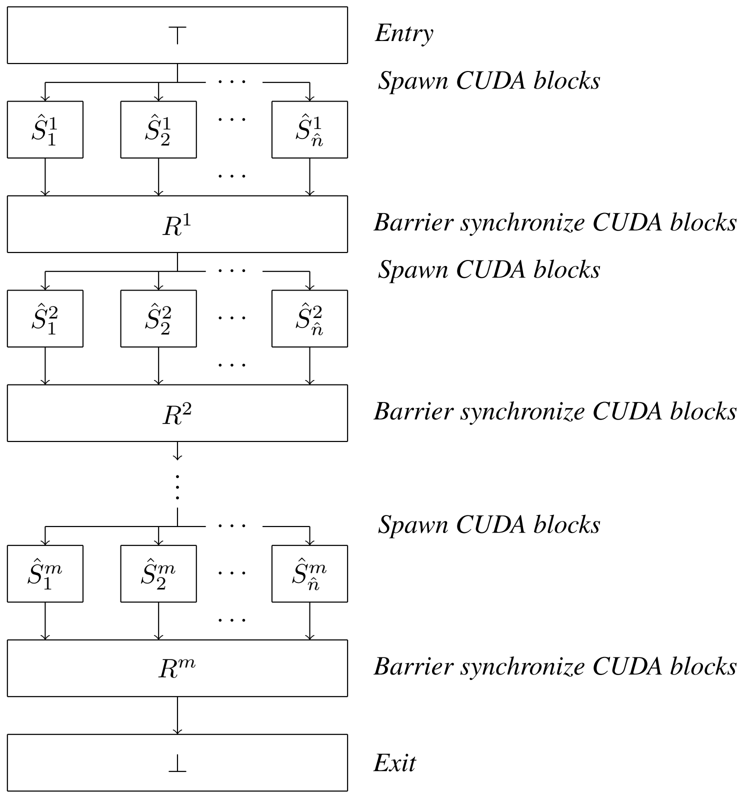


Figure 3.1: Showing basic parallelization model of Diderot strands using CUDA. The program entrypoint is shown as $\top$ and the exit point as $\bot$.

With the control flow and threads managed, the other main challenge is how to manage data. Diderot has a few flavors of data:

1. Strand-local data
2. Strand statuses
3. Global data
4. Metadata of strand collection

Diderot stores strand-local data in a large strand array. Given that we restricted our computation model not to spawn new threads, we only need to allocate all the data once, as we map multiple strands to a single thread (in different ways, depending on the strategy). The only problem here is that if the required memory for strand-local data exceeds GPU VRAM on a target GPU. Luckily, CUDA unified memory offers the solution to this problem. Hence, we allocate strand memory through CUDA unified memory by default (though there is an option to not use unified memory). As CUDA unified memory usually incurs a performance penalty, we evaluate the performance impact in Section 4.2. Owning to the predictable nature of access patterns, we can give CUDA hints to optimize the sharing of memory between CPU and GPU, to mitigate these effects. Like with the other strand-local data, each strand has a status. Statuses indicate whether a strand is still active and whether it terminates successfully. Like with other strand-local data, the status is stored in a large fixed-size array.

Global data is, in general, a bit more complicated, as it is a nested structure of simple data types (such as integers, floats, etc.) and composite data types: structures and fixed/variable-sized arrays. Hence, we need to deep-copy data that was prepared on the CPU to the GPU. We do this using CUDA unified memory as this allows us to load this data on demand.

Next to data associated with individual strands, we also keep metadata on the entire strand array, such as how many strands are alive. As this metadata is frequently accessed in supersteps and reduction steps, we use CUDA unified

Benchmark name	Number of strands	μ	σ	CV
illust-vr	307 200	775.904	378.906	0.488
lic2d	572 220	9.975	0.474	0.048
mandelbrot	4 000 000	284.571	679.472	2.388
ridge3d	1 728 000	8.155	5.939	0.728
vr-lite-cam	165 600	457.903	311.090	0.679

Figure 3.2: A table showing the average (μ), standard deviation (σ) and correlation value (CV $= {}^{\sigma}/_{\mu}$) of number of steps run for each strand.

memory with access pattern hints to store this data. Luckily the entire set of metadata is less than 150 bytes, so copying does not cause much overhead. Some updates of the metadata, however, such as counting the number of active strands, require atomic operations, which tend to have a significant impact on performance. We discuss in later sections the performance impact this has in practice.

Lastly, we need to handle a variety of GPUs and systems with multiple GPUs. As alluded to before, we utilize CUDA unified memory to solve the memory overprovisioning challenge. Though, there are some additional limitations for different GPUs. Older CUDA-compatible GPUs do not support unified memory and for those GPUs overprovisioning is not available but we are providing an implementation without unified memory. Moreover, different GPUs allow different sizes for both grids and blocks. As we expect different behavior for grid and block sizes depending on the GPU we left them parameterized with sensible default values. Moreover, in systems with multiple GPUs this problem becomes more complicated as GPUs with different features are hard to choose from. We choose the heuristic of picking the GPU with the most VRAM by default but allowing the user to specify which GPU to run on or provide an interactive command line choice system.

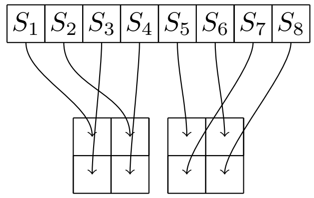
3.2 Index space permutation

When assigning strands to positions in the CUDA grid in blocks, our initial strategy is to map threads linearly into the nested grid provided by CUDA. We show an example of this mapping in Figure 3.3a. Once assigned, the strands stay in their place until the entire computation ends. If a strand dies or stabilizes, its stepping becomes a no-op. Suppose, however, an entire block of strands consists entirely of no-op strands. In that case, the computation causes a load imbalance, leaving some blocks workless, while other blocks are still computing causing overall inefficiencies. In Diderot's programming model, we believe that in practice strands with similar ids behave similarly and hence terminate at similar times. Empirical findings confirm this for our set of benchmarks (which introduce further in Section 4.1.1). Figure 3.2 shows that many benchmarks have high correlation values (CVs) for the number of steps run across different strands. We believe that such benchmarks will benefit the most from index space permutation.

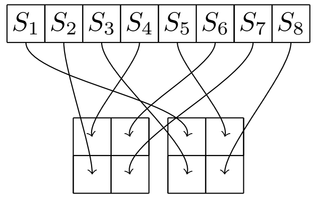
To avoid real-world load imbalance, we propose randomizing the assignment of strands to positions in the nested CUDA grid. We show an example of this process in Figure 3.3b. The advantage is that on average we should see approximately equal load distribution. Similarly, with the underlying assumption that strands with similar indices behave similarly, their data access patterns might be very similar. With the index space permutation optimization, this might also mean that we lose the advantages of cache effects that come from similar data access patterns. We analyze this in more detail in Section 4.2.

3.3 Batching

A third approach for parallelization that we call batching, works by optimizing Diderot programs that do not have any reduction steps. We show how the original model would look in this scenario in Figure 3.4 and the pseudo-code in Algorithm 2. This approach effectively runs one step of each strand assigned to one thread. The batching approach fundamentally does something similar, except that it keeps running a single thread to completion and then switching to the next thread. The goal with this model is to improve performance by keeping data on threads cached for better cache performance. We also expect less swapping overhead in the case of GPU memory overprovisioning, as each time a thread is switched it might have to be fetched from system memory instead of being on GPU.



(a) Linearly mapping eight strands into a CUDA grid with 2 blocks, each sized 2×2.



(b) Randomly mapping eight strands into a CUDA grid with 2 blocks, each sized 2×2.

Figure 3.3: Visualization strategies of mapping strands into a CUDA grid

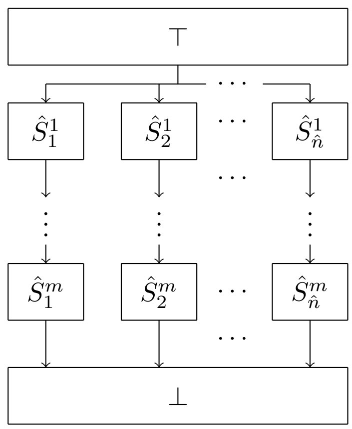


Figure 3.4: Showing basic parallelization model of Diderot strands using CUDA without reduction. The program entrypoint is shown as $\top$ and the exit point as $\perp$.

3.4 Global Queue

Next to the basic and batching approaches, we have another approach that aims to mostly prevent waiting for other strands to complete a step and reduce scheduling overhead. This approach is restricted to Diderot programs that do not have reduction steps or strand-to-strand communication. If we recall the basic approach, we see that without reduction steps, strands run in the same place until the last strand finishes. We show this in Figure 3.4. As discussed in Section 3.2, however, there can be discrepancies in the number of steps between strands causing some CUDA threads to idle. The approach we choose is to queue up all strands, and then let each CUDA thread take c strands out of the said queue at a time. The CUDA thread then runs the strand until completion without interruption. Formally we consider $\bar{S}_j := \{S_{j\cdot c}, \ldots, S_{\min(n,(j+1)\cdot c-1)}\}$ a group of c strands and n_{thread} the number of CUDA threads. Shown in Figure 3.6, we take such groups $\bar{S}$ are run to complete them as they become available. This means strand groups can have heterogenous run times, yet all CUDA threads always have work until all ($\bar{n}$) groups of strands have been assigned. With this system there are no explicit waiting periods of CUDA threads except at the end waiting for the completion of the last batch of work. The assignment of such batches works by having a global counter j and when a CUDA thread has no work assigned it atomically increments j and take $\bar{S}_j$ as its next task batch. Through this implementation, we do not need to implement an actual queue data structure on the GPU, but instead, reuse our previously constructed and copied strand array. We show our pseudo-code implementation in Algorithm 3.

Therefore, we do not need to alter our memory allocation and copy patterns. Atomic operations behave poorly in GPUs, however, if not done carefully: When multiple threads try to increment a counter at the same time performance

Algorithm 2 Batching CUDA implementation

function BATCHCUDAIMPLEMENTATION(nStrands, nCudaThreads)
    blockSize $\leftarrow$ nStrands / nCudaThreads
    for all i $\in$ [0...nCudaThreads) **do**
        RunThreads(i, blockSize)
    end for
    Assert no strands are alive
end function
function RUNTHREAD(threadId, blockSize)
    for all i $\in$ [threadId * blockSize, ..., threadId * (blockSize + 1)) **do**
        while Strand i is alive **do**
            Update i^{th} strand
        end while
    end for
end function

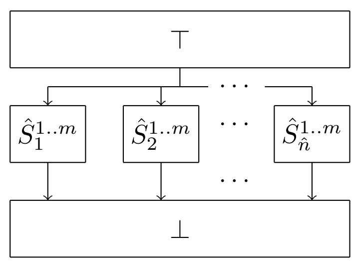


Figure 3.5: Showing the batching parallelization model of Diderot strands using CUDA. This assumes no reduction steps. The program entrypoint is shown as $\top$ and the exit point as $\perp$.

is lost. Given the number of concurrent threads in theory this can cause a lot of simultaneous waiting. Our conjecture is that variation in strand batches' runtime and hence there will not be too many slowdowns due to atomic operations. At the start of the computation, however, all threads need work assigned, therefore atomic operations would impede performance. As a mitigation, we pre-assign batches before initialization to threads and only perform atomic operations for dequeuing after that.

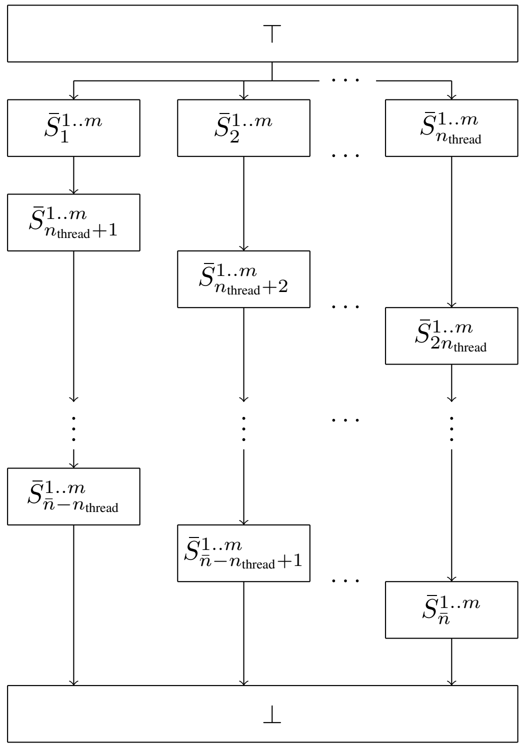


Figure 3.6: Showing the global queue parallelization model of Diderot strands using CUDA. This assumes no reduction steps. The program entrypoint is shown as $\top$ and the exit point as $\perp$.

Algorithm 3 Batching CUDA implementation

```
function GLOBALQUEUEIMPLEMENTATION(nStrands, nCudaThreads)
    GlobalIndex = nCudaThreads
    for all i ∈ [0...nStrands) do
        RunThreads(i, nStrands, Ref(GlobalIndex))
    end for
    Assert no strands are alive
end function
function RUNTHREAD(threadId, nStrands, Ref(GlobalIndex)) i ← threadId        ▷ Initial strand assignment
    while i < nStrands do
        while Strand i is alive do
            Update i^th strand
        end while
        i ← AtomicAdd(Ref(GlobalIndex), 1)
    end while
end function
```

Chapter 4

Evaluation

4.1 Methodology

4.1.1 Benchmarking

To evaluate our work we benchmark our CUDA implementations, against CPU sequential and parallel compute. For all of our comparisons, we use system with two Intel Xeon Gold 6142 CPUs with 16 cores (32 threads) clocked at 2.60GHz, 96GiB of DDR4 ECC system memory, and an NVIDIA Tesla V100 16GB GPU. We keep the system constant and ensure there is no other load for the duration of the benchmark.

To have comparable numbers we use the benchmark set from earlier Diderot work [16, 6]:

- `illust-vr`: A volume renderer utilizing tensor expressions [17].
- `lic2d`: Line integral convolution visualization of a synthetic 2D vector field [4].
- `mandelbrot`: Mandelbrot fractal rendering [19].
- `ridge3d`: A ridge detection algorithm applied on 3D medical images [10].
- `vr-lite-cam`: A phong-shading based volume renderer [21].

As we implement three different parallelization strategies we compare them against each other. Given that all the benchmarks above do not have strand-to-strand communication or reduction steps, we can compare the general (see Section 3.1), batching (see Section 3.3), and global queue (see Section 3.4) approaches. We also test the general and batching approaches with the index space permutation (see Section 3.2). Moreover, we test the general and global queue approach with and without CUDA unified memory.

While evaluating approaches we also test different parameters for the grid size and work unit size for global queuing. We test $n = 64, 128, \ldots, 1024$ and $b = 64, 128, \ldots, b_{\text{max}}$, where b_{max} is the maximum size of b for the overall CUDA grid size to be valid on our GPU. For work unit size c we test $1, 2, 4, 8, 16$.

To gain statistically reliable results we run each benchmark 50 times and report the mean and standard deviation of our results. We discuss the results in Section 4.2 and have the raw data in Appendix A.

4.1.2 Analysis

We evaluate the speed-up of the CUDA versions of the benchmark over the sequential implementation. We use the sequential version as a baseline to get accurate numbers to compare different GPU implementations and assign global meaning to check the overall viability of the CUDA implementation. Our focus will be on the difference between our strategies, however, as we aim to compare them and their respective performance. To recap, we will compare the following strategies:

- General strategy (see Section 3.1), with index space permutation (see Section 3.2), with CUDA unified memory, and without either of these additions

- Batching strategy (see Section 3.3), with and without index space permutation (see Section 3.2)
- Global queue strategy (see Section 3.4), with and without CUDA unified memory

Further, we analyze our results to evaluate the effectiveness by running the fastest set of parameters for each feature set and each benchmark with a profiler. We evaluate how much time is spent copying data and synchronizing. This gives us metrics as to what our generated programs spend the most time on. Seeing that our programs spend the most time on computation instead of data copying or synchronization tells us that we have parallelized effectively. From previous work, we know that the compilation of complex operations in Diderot beats hand-optimized code. Therefore, we can assume that a high share of computation time spent on actual strand steps can be seen as an indicator for efficient computation.

4.2 Results

In this section, we look at benchmark results from our work. We describe the methodology in Section 4.1.1. All of our results can be found in detail in tables in Appendix A. We present the results of each strategy in Figure 4.1.

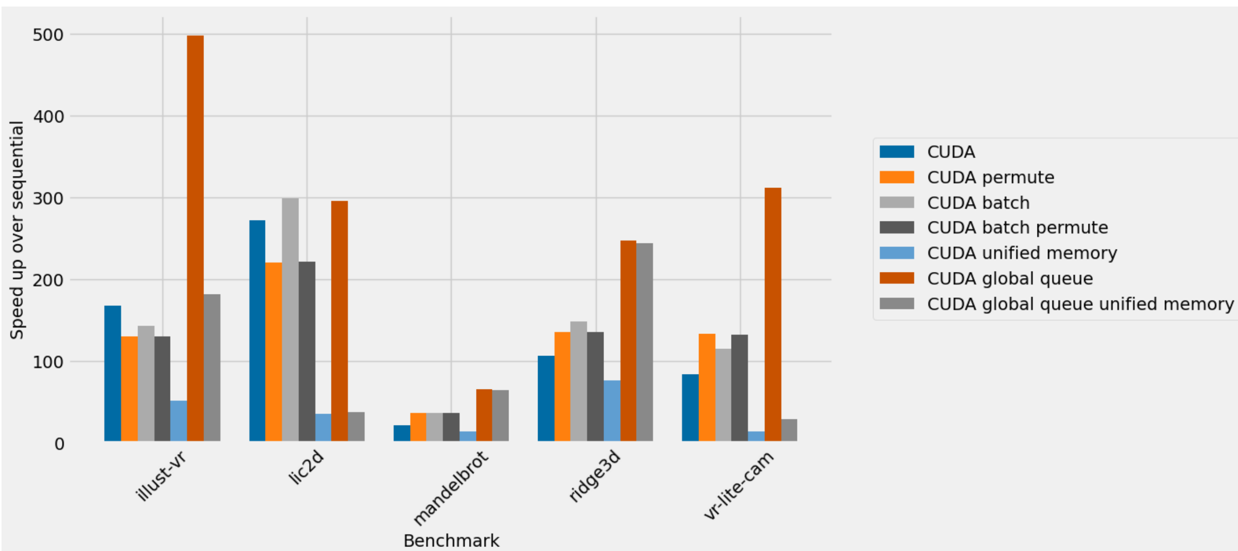


Figure 4.1: Results of each parallelization strategy on each benchmark with optimal parameter choice

4.2.1 Parallelization strategies

Over the various benchmarks we see clear patterns emerge:

1. Global queue computation is always the fastest strategy
2. Permutation does not aid performance but rather hurts it
3. There are clear trends with the parameters (which we explore in Section 4.2.2)
4. CUDA unified memory is slower than the hand-optimized memory but usually not significantly (we explore this in Section 4.2.3)

Let us explore the first two observations:

Our benchmarks all fall within the category of Diderot programs that do not require mid-run synchronization. Hence, we expect global queue computation to be the fastest strategy if the synchronization for new work items to be allocated does not cause too much overhead. Previously, we hypothesized that the synchronization overhead will be minimal due to differences in thread runtimes. Our results confirm this hypothesis: The global queue strategy consistently achieves the overall fastest results, except in lic2d. lic2d has runtimes shorter than 0.1 seconds for all strategies and hence setup overhead of the queue outweighs the benefits. Given that we target algorithms with significant runtimes, we consider this not significant. It outperforms batching by up to 4x and the base strategy by up to 3x. Note that we look at the fastest results for each category. Notice that batching also outperforms the base strategy by due to the lower overhead coming from the fewer synchronization points. Batching underperforms the base strategy in illust-vr. This is likely due to the high number of steps that are similar across threads and the resulting cache-locality in the default implementation.

The other observation we make is that permutation of threads' advantageousness is related to the correlation value of the number of strand steps runs for a given benchmark, as seen in Figure 3.2. On benchmarks with higher CVs (such as mandelbrot), it outperforms the default strategy by quite a bit. Whereas on benchmarks with lower CVs, such as ridge3d or vr-lite-cam, the performance increase is smaller. On benchmarks with even smaller CVs, the performance then drops compared to the default strategy. We see a similar pattern with perturbations on the batching strategy, however, the CV needs to be higher for the permutation to be advantageous.

While not overly significant there is no advantage to using permutation and on the contrary, it causes more code complication.

Looking at the comparison to the sequential version of Diderot, we see the global queue strategy achieving speedups from 65 to 500 times. The differences between benchmarks likely come down to overall runtime, where lower runtimes incur a larger penalty from spawning GPU work. For example, mandelbrot only takes 0.09sec to run in the best GPU run, which means that large amounts of the runtime are spent setting up the environment and copying data. The profiler confirms this suspicion.

4.2.2 Parameter comparison

For all CUDA parallelization strategies, we notice that selecting the largest n and b is either optimal or in the cases where it is not true are close to the optimal run. We clearly see that no benchmark is bottlenecked by having too few strands for the parallel processors, as all benchmarks have more strands than CUDA threads, even at maximum grid size (see Figure 3.2). This is what we expected, as selecting the large parameters allows for maximal parallelism and within our model, this should yield advantages. This also shows us that our implementation scales well up to a certain point. Given the similar computation model but different optimal parameters, it is likely dependent on the problem size of given benchmarks or simple statistical noise. Up to the local optima our implementation scales linearly, showing that our overhead from memory operations is low. Attaching a profiler confirms this hypothesis and shows that actual computation makes up over 98%. For the global queue strategy, we see that a parameter of $c = 1$ is always optimal. This is likely because fetching new tasks is inexpensive and contrary to our previous worries synchronization is minimal. Hence, the trading of granularity for synchronization seems like the optimal strategy.

4.2.3 CUDA unified memory

Furthermore, we see that using CUDA unified memory incurs quite a hefty performance penalty. We see that this penalty is consistent across parallelization models and benchmarks, except ridge3d and mandelbrot, which have a very high number of strands being run at once (see Figure 3.2). In general, we see the more strands being run the less the unified memory overhead becomes. This is likely as with the higher copying requirements of strand data, the initial overhead of creating the unified address space is amortized. Unified memory vastly simplifies implementation and maintainability and also allows for overprovisioning, but we must conclude that it is not advantageous for our implementation. We also pose the conjecture that the hand-optimization benefits of not using unified memory outweigh the ease of development and additional memory safety that unified memory provides in most cases.

Chapter 5

(Possible) Future work

In this section, we present multiple possibilities to build on the promising results that we achieved. Broadly, one can group the advancements in two categories: Supporting more Diderot features and exploiting other possibilities of CUDA. Throughout this work we mention a few assumptions that restrict the kinds of programs that can automatically be translated to CUDA. One of the largest assumptions is that we do not support dynamic strand creation. Given our results, and experience with CUDA, we believe that the most effective way to implement this feature is to generalize the global queue. This means that each strand would be enqueued, runs for one step, followed by a global synchronization and global operation. While the global operation is happening, additional strand memory would be allocated and initialized. The new strands would simply be added to the queue. With the dynamic paging of CUDA unified memory, we believe this can be elegantly implemented.

The work on dynamic thread creation could also work well with dual state, as the modification to a strand array could be easily amended with copy operations to store the previous state. This would allow for strand queries and previous data to be easily retrieved. A potential challenge though is the constrained memory that a GPU has. Doubling the memory required with two copies of state can easily lead to issues. One solution could be to utilize CUDA unified memory's automatic paging with smart access patterns hints based on the expected behavior.

Another feature of Diderot to be supported is dynamic sequences. Dynamic sequences are variably sized lists that can be part of strand state or global state. This is a bit harder to integrate into the existing system due to the limited memory management capabilities and strict space constraints on GPUs. There are some libraries, such as Thrust [2]. to address this problem but they often don't work well with CUDA unified memory and hence logic for swapping data has to be effectively implemented. As an optimization for global state dynamic sequences, one could use a regular vector on the CPU side to operate on data and copy the changed data into a fixed array on the GPU, as the global sequences become read-only for strand updates. Optimizations like these will likely create the need to use something more advanced than existing libraries.

Chapter 6

Conclusion

In this work, we looked at adding automatic parallelization for CUDA targets to the scientific analysis DSL Diderot. Recall, that we devised three strategies: one default strategy that closely follows the BSP model by parallelizing the update steps. Another is for programs without a global update step, we simply run strands to termination but keep the assignment from CUDA threads to strands static. And lastly, iterating on the previous idea, where rather than statically assigning strands to CUDA threads, we put all strands in a queue and have CUDA cores work on them in small chunks. To address strand divergence issues we also implemented a system where the index space of strands and hence the assignment to CUDA cores is stochastic.

In our evaluation, we saw that we see great performance gains on any program where the actual execution time is non-negligible. We also see that CUDA unified memory introduces quite high overhead for benchmarks that do not have a very high number of strands. Even for benchmarks where CUDA unified memory performs better, it still lags the performance without unified memory. Permutation, on the other hand, is not clearly better or worse than not having the mechanism. It highly depends on the workload: Workloads where the load per strand is highly uneven, have divergence issues without permutation. In these cases, permutation benefits performance. We advocate for turning it on by default but allowing developers to specify not using it.

In general, our findings lead us to conclude that when dealing with DSLs designed for parallelism, adding GPU support is a viable option. We also believe that utilizing CUDA unified memory is in many cases a costly trade-off due to the high overhead for a simpler implementation. We hope to see future work build on our progress to fully bring Diderot's features onto the GPU. Our results have shown this is a promising area of work and we believe it comes with many interesting challenges.

Appendix A

Raw Data

A.1 Illust-vr

Table A.1: Results of benchmarking illust-vr using execution method "sequential"

mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
31.614	**0.148**	**31.550**	**1.000x**

Table A.2: Results of benchmarking illust-vr using execution method "cuda"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	4.351	0.002	4.288	7.266x
16	32	2.882	0.003	2.818	10.971x
16	64	1.951	0.006	1.888	16.203x
16	128	1.287	0.003	1.223	24.566x
16	256	0.925	0.004	0.862	34.169x
32	16	2.446	0.003	2.382	12.927x
32	32	1.710	0.004	1.646	18.490x
32	64	1.038	0.003	0.975	30.451x
32	128	0.672	0.002	0.608	47.058x
32	256	0.490	0.001	0.427	64.490x
64	16	1.334	0.002	1.271	23.698x
64	32	0.881	0.002	0.818	35.877x
64	64	0.524	0.002	0.461	60.304x
64	128	0.366	0.001	0.303	86.314x
64	256	0.280	0.001	0.217	112.863x
64	1024	0.190	0.002	0.126	166.517x
128	16	0.758	0.001	0.695	41.706x
128	32	0.508	0.001	0.444	62.268x

Continued on next page

Table A.2: Results of benchmarking illust-vr using execution method "cuda" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	64	0.354	0.000	0.290	89.403x
128	128	0.266	0.000	0.202	118.919x
128	256	0.229	0.001	0.165	138.240x
128	**512**	**0.189**	**0.000**	**0.126**	**166.834x**
256	16	0.471	0.001	0.408	67.081x
256	32	0.342	0.001	0.278	92.556x
256	64	0.263	0.000	0.200	120.050x
256	128	0.228	0.001	0.164	138.846x
256	256	0.190	0.000	0.126	166.720x
512	16	0.296	0.000	0.232	106.973x
512	32	0.249	0.000	0.186	126.830x
512	64	0.231	0.001	0.167	137.143x
512	128	0.190	0.000	0.126	166.764x
1024	16	0.266	0.001	0.202	118.906x
1024	32	0.221	0.000	0.157	143.187x
1024	64	0.190	0.000	0.126	166.717x

Table A.3: Results of benchmarking illust-vr using execution method "cuda-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	3.266	0.003	3.203	9.679x
16	32	2.008	0.005	1.945	15.744x
16	64	1.345	0.007	1.281	23.511x
16	128	1.057	0.003	0.993	29.919x
16	256	0.800	0.002	0.737	39.509x
32	16	1.628	0.003	1.564	19.422x
32	32	1.007	0.004	0.944	31.381x
32	64	0.688	0.003	0.625	45.940x
32	128	0.547	0.002	0.483	57.841x
32	256	0.455	0.001	0.391	69.536x
64	16	0.813	0.002	0.750	38.878x
64	32	0.505	0.002	0.441	62.634x
64	64	0.356	0.002	0.292	88.875x
64	128	0.338	0.002	0.274	93.576x
64	256	0.330	0.002	0.266	95.904x
128	16	0.469	0.003	0.405	67.415x

Continued on next page

Table A.3: Results of benchmarking illust-vr using execution method "cuda-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	32	0.347	0.001	0.284	91.102x
128	64	0.305	0.001	0.242	103.493x
128	128	0.294	0.002	0.230	107.692x
128	256	0.289	0.003	0.225	109.402x
256	16	0.343	0.004	0.279	92.286x
256	32	0.294	0.001	0.231	107.476x
256	64	0.287	0.002	0.224	110.017x
256	128	0.282	0.003	0.218	112.110x
512	**16**	**0.244**	**0.002**	**0.181**	**129.547x**
512	32	0.281	0.001	0.218	112.449x
512	64	0.266	0.003	0.202	118.901x
1024	16	0.267	0.003	0.203	118.443x
1024	32	0.262	0.003	0.199	120.527x

Table A.4: Results of benchmarking illust-vr using execution method "cuda-batch"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	3.157	0.003	3.094	10.013x
16	32	1.828	0.002	1.764	17.298x
16	64	1.473	0.003	1.409	21.468x
16	128	1.142	0.002	1.079	27.680x
16	256	0.876	0.001	0.812	36.099x
32	16	1.661	0.003	1.598	19.028x
32	32	1.070	0.001	1.007	29.544x
32	64	0.708	0.002	0.644	44.678x
32	128	0.555	0.002	0.491	56.973x
32	256	0.492	0.000	0.429	64.220x
64	16	0.821	0.003	0.758	38.487x
64	32	0.515	0.001	0.452	61.334x
64	64	0.344	0.001	0.281	91.834x
64	128	0.324	0.001	0.260	97.666x
64	256	0.343	0.001	0.279	92.171x
128	16	0.471	0.002	0.408	67.070x
128	32	0.325	0.001	0.261	97.388x
128	64	0.278	0.001	0.214	113.736x
128	128	0.298	0.001	0.234	106.240x

Continued on next page

Table A.4: Results of benchmarking illust-vr using execution method "cuda-batch" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	256	0.239	0.003	0.176	132.127x
256	16	0.354	0.003	0.290	89.383x
256	32	0.272	0.000	0.209	116.083x
256	64	0.303	0.001	0.240	104.298x
256	128	0.229	0.003	0.166	137.859x
512	16	0.259	0.001	0.195	122.203x
512	32	0.292	0.001	0.229	108.190x
512	64	0.223	0.002	0.160	141.684x
1024	16	0.268	0.003	0.205	117.806x
1024	**32**	**0.222**	**0.003**	**0.158**	**142.711x**

Table A.5: Results of benchmarking illust-vr using execution method "cuda-batch-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	3.267	0.004	3.203	9.677x
16	32	2.007	0.006	1.943	15.755x
16	64	1.341	0.006	1.278	23.569x
16	128	1.057	0.003	0.993	29.910x
16	256	0.799	0.002	0.736	39.545x
32	16	1.628	0.003	1.565	19.415x
32	32	1.007	0.004	0.943	31.395x
32	64	0.688	0.003	0.625	45.941x
32	128	0.548	0.002	0.484	57.740x
32	256	0.455	0.002	0.391	69.541x
64	16	0.813	0.002	0.750	38.868x
64	32	0.504	0.002	0.441	62.673x
64	64	0.356	0.002	0.292	88.845x
64	128	0.338	0.002	0.274	93.596x
64	256	0.329	0.002	0.266	96.072x
128	16	0.469	0.004	0.406	67.370x
128	32	0.347	0.002	0.283	91.226x
128	64	0.305	0.001	0.242	103.574x
128	128	0.294	0.002	0.230	107.599x
128	256	0.289	0.003	0.226	109.226x
256	16	0.343	0.003	0.279	92.208x
256	32	0.294	0.001	0.231	107.422x

Continued on next page

Table A.5: Results of benchmarking illust-vr using execution method "cuda-batch-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	64	0.287	0.002	0.223	110.181x
256	128	0.282	0.003	0.219	111.910x
512	**16**	**0.244**	**0.002**	**0.180**	**129.624x**
512	32	0.282	0.001	0.218	112.197x
512	64	0.267	0.003	0.203	118.626x
1024	16	0.266	0.002	0.203	118.706x
1024	32	0.262	0.003	0.199	120.659x

Table A.6: Results of benchmarking illust-vr using execution method "cuda-unified-memory"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	4.470	0.006	4.406	7.073x
16	32	3.096	0.013	3.033	10.211x
16	64	2.221	0.036	2.158	14.233x
16	128	1.538	0.028	1.474	20.560x
16	256	1.175	0.020	1.111	26.914x
32	16	2.621	0.009	2.557	12.064x
32	32	2.002	0.049	1.938	15.791x
32	64	1.406	0.071	1.342	22.488x
32	128	1.001	0.041	0.937	31.595x
32	256	0.811	0.034	0.747	38.985x
64	16	1.504	0.011	1.440	21.022x
64	32	1.193	0.058	1.129	26.504x
64	64	0.894	0.067	0.831	35.348x
64	128	0.859	0.068	0.796	36.793x
64	256	0.717	0.041	0.653	44.096x
128	16	1.101	0.052	1.037	28.715x
128	32	0.999	0.089	0.936	31.633x
128	64	1.106	0.079	1.042	28.593x
128	128	0.926	0.050	0.863	34.122x
128	**256**	**0.616**	**0.037**	**0.552**	**51.324x**
256	16	1.185	0.121	1.122	26.670x
256	32	1.215	0.088	1.152	26.012x
256	64	1.222	0.024	1.159	25.865x
256	128	0.823	0.041	0.759	38.422x

Continued on next page

Table A.6: Results of benchmarking illust-vr using execution method "cuda-unified-memory" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	16	1.159	0.103	1.095	27.288x
512	32	1.266	0.032	1.202	24.978x
512	64	0.990	0.028	0.926	31.942x
1024	16	0.900	0.072	0.836	35.141x
1024	32	0.823	0.045	0.760	38.413x

Table A.7: Results of benchmarking illust-vr using execution method "cuda-global-queue"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	2.153	0.003	2.090	14.684x
16	16	2	2.308	0.000	2.244	13.700x
16	16	4	2.581	0.000	2.518	12.247x
16	16	8	2.715	0.001	2.651	11.646x
16	16	16	2.962	0.002	2.898	10.674x
16	32	1	1.178	0.000	1.115	26.830x
16	32	2	1.288	0.000	1.224	24.554x
16	32	4	1.557	0.000	1.494	20.300x
16	32	8	1.709	0.001	1.646	18.495x
16	32	16	1.932	0.002	1.868	16.367x
16	64	1	0.617	0.000	0.554	51.202x
16	64	2	0.721	0.000	0.658	43.832x
16	64	4	0.942	0.001	0.878	33.564x
16	64	8	1.096	0.002	1.032	28.855x
16	64	16	1.325	0.005	1.261	23.866x
16	128	1	0.364	0.000	0.300	86.968x
16	128	2	0.496	0.001	0.432	63.796x
16	128	4	0.750	0.001	0.687	42.144x
16	128	8	0.915	0.003	0.852	34.537x
16	128	16	1.089	0.003	1.025	29.037x
16	256	1	0.266	0.000	0.202	119.061x
16	256	2	0.417	0.001	0.354	75.779x
16	256	4	0.640	0.001	0.577	49.373x
16	256	8	0.726	0.003	0.663	43.540x
16	256	16	0.805	0.003	0.741	39.289x
32	16	1	1.066	0.003	1.002	29.663x

Continued on next page

Table A.7: Results of benchmarking illust-vr using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	16	2	1.142	0.000	1.078	27.684x
32	16	4	1.278	0.000	1.215	24.733x
32	16	8	1.345	0.001	1.282	23.497x
32	16	16	1.470	0.002	1.406	21.512x
32	32	1	0.585	0.000	0.521	54.086x
32	32	2	0.640	0.000	0.576	49.410x
32	32	4	0.775	0.000	0.711	40.809x
32	32	8	0.851	0.001	0.788	37.129x
32	32	16	0.965	0.001	0.901	32.766x
32	64	1	0.307	0.001	0.244	102.816x
32	64	2	0.360	0.000	0.296	87.888x
32	64	4	0.470	0.001	0.407	67.195x
32	64	8	0.548	0.001	0.485	57.672x
32	64	16	0.667	0.002	0.603	47.401x
32	128	1	0.182	0.000	0.118	173.899x
32	128	2	0.248	0.000	0.184	127.679x
32	128	4	0.375	0.001	0.312	84.278x
32	128	8	0.461	0.002	0.397	68.631x
32	128	16	0.567	0.003	0.504	55.737x
32	256	1	0.134	0.000	0.071	235.059x
32	256	2	0.210	0.001	0.147	150.211x
32	256	4	0.325	0.001	0.261	97.307x
32	256	8	0.396	0.003	0.332	79.905x
32	256	16	0.508	0.004	0.445	62.229x
64	16	1	0.532	0.003	0.469	59.404x
64	16	2	0.572	0.000	0.509	55.252x
64	16	4	0.638	0.000	0.575	49.527x
64	16	8	0.673	0.001	0.609	46.991x
64	16	16	0.739	0.001	0.676	42.763x
64	32	1	0.298	0.000	0.234	106.196x
64	32	2	0.323	0.000	0.260	97.756x
64	32	4	0.391	0.000	0.327	80.861x
64	32	8	0.432	0.001	0.368	73.224x
64	32	16	0.497	0.001	0.434	63.564x
64	64	1	0.158	0.000	0.095	199.691x
64	64	2	0.184	0.000	0.121	171.685x

Continued on next page

Table A.7: Results of benchmarking illust-vr using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	64	4	0.240	0.000	0.177	131.599x
64	64	8	0.281	0.001	0.218	112.315x
64	64	16	0.357	0.002	0.293	88.630x
64	128	1	0.097	0.000	0.034	325.204x
64	128	2	0.131	0.000	0.067	242.050x
64	128	4	0.194	0.001	0.131	162.632x
64	128	8	0.254	0.002	0.190	124.523x
64	128	16	0.363	0.003	0.299	87.093x
64	256	1	0.076	0.000	0.013	415.922x
64	256	2	0.115	0.000	0.052	273.774x
64	256	4	0.188	0.002	0.125	168.110x
64	256	8	0.272	0.005	0.208	116.255x
64	256	16	0.395	0.008	0.331	80.079x
128	16	1	0.272	0.002	0.209	116.208x
128	16	2	0.294	0.000	0.231	107.423x
128	16	4	0.337	0.000	0.274	93.806x
128	16	8	0.362	0.001	0.298	87.394x
128	16	16	0.415	0.001	0.351	76.181x
128	32	1	0.156	0.000	0.092	202.729x
128	32	2	0.178	0.000	0.115	177.166x
128	32	4	0.228	0.000	0.164	138.672x
128	32	8	0.262	0.001	0.199	120.471x
128	32	16	0.339	0.002	0.276	93.257x
128	64	1	0.093	0.000	0.029	340.397x
128	64	2	0.119	0.000	0.056	265.239x
128	64	4	0.171	0.001	0.108	184.612x
128	64	8	0.222	0.002	0.158	142.412x
128	64	16	0.308	0.016	0.245	102.540x
128	128	1	0.067	0.000	0.004	469.211x
128	128	2	0.099	0.000	0.036	317.836x
128	128	4	0.157	0.001	0.093	201.650x
128	128	8	0.210	0.003	0.147	150.483x
128	128	16	0.334	0.002	0.271	94.557x
128	256	1	0.066	0.000	0.002	482.440x
128	256	2	0.099	0.000	0.036	318.662x
128	256	4	0.163	0.001	0.099	194.106x

Continued on next page

Table A.7: Results of benchmarking illust-vr using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	256	8	0.232	0.002	0.168	136.317x
128	256	16	0.323	0.001	0.260	97.837x
256	16	1	0.144	0.002	0.081	219.328x
256	16	2	0.162	0.000	0.099	194.644x
256	16	4	0.195	0.000	0.132	161.854x
256	16	8	0.224	0.001	0.161	140.857x
256	16	16	0.288	0.004	0.224	109.918x
256	32	1	0.091	0.000	0.027	347.554x
256	32	2	0.115	0.000	0.052	274.682x
256	32	4	0.163	0.001	0.099	194.211x
256	32	8	0.216	0.002	0.153	146.362x
256	32	16	0.300	0.002	0.236	105.468x
256	64	1	0.065	0.000	0.002	483.750x
256	64	2	0.096	0.001	0.033	328.932x
256	64	4	0.156	0.001	0.093	202.416x
256	64	8	0.210	0.002	0.147	150.417x
256	64	16	0.351	0.004	0.288	90.003x
256	128	1	0.064	0.000	0.000	495.024x
256	128	2	0.100	0.001	0.036	316.750x
256	128	4	0.156	0.001	0.092	202.763x
256	128	8	0.218	0.005	0.155	144.757x
256	128	16	0.298	0.001	0.234	106.163x
512	16	1	0.085	0.001	0.022	370.529x
512	16	2	0.107	0.000	0.044	295.441x
512	16	4	0.142	0.001	0.079	222.325x
512	16	8	0.169	0.001	0.105	187.410x
512	16	16	0.244	0.002	0.180	129.778x
512	32	1	0.065	0.000	0.002	485.403x
512	32	2	0.095	0.000	0.031	333.780x
512	32	4	0.147	0.001	0.084	214.776x
512	32	8	0.209	0.001	0.146	151.015x
512	32	16	0.365	0.003	0.302	86.557x
512	64	1	0.064	0.000	0.000	497.646x
512	64	2	0.099	0.001	0.035	319.758x
512	64	4	0.156	0.002	0.093	202.078x
512	64	8	0.210	0.006	0.146	150.714x

Continued on next page

Table A.7: Results of benchmarking illust-vr using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	64	16	0.299	0.001	0.236	105.681x
1024	16	1	0.077	0.001	0.013	410.855x
1024	16	2	0.102	0.000	0.038	310.539x
1024	16	4	0.143	0.001	0.080	220.508x
1024	16	8	0.174	0.002	0.111	181.424x
1024	16	16	0.247	0.002	0.183	128.038x
1024	**32**	**1**	**0.063**	**0.000**	**0.000**	**497.881x**
1024	32	2	0.098	0.001	0.035	322.232x
1024	32	4	0.155	0.002	0.091	204.075x
1024	32	8	0.210	0.002	0.146	150.569x
1024	32	16	0.303	0.001	0.240	104.321x

Table A.8: Results of benchmarking illust-vr using execution method "cuda-global-queue-unified-memory"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	2.167	0.002	2.103	14.590x
16	16	2	2.324	0.001	2.261	13.602x
16	16	4	2.601	0.001	2.538	12.153x
16	16	8	2.737	0.001	2.674	11.549x
16	16	16	3.002	0.003	2.938	10.532x
16	32	1	1.192	0.001	1.129	26.516x
16	32	2	1.301	0.001	1.238	24.297x
16	32	4	1.574	0.001	1.511	20.081x
16	32	8	1.744	0.003	1.680	18.130x
16	32	16	2.045	0.012	1.981	15.460x
16	64	1	0.638	0.002	0.574	49.580x
16	64	2	0.740	0.002	0.677	42.694x
16	64	4	0.975	0.002	0.911	32.438x
16	64	8	1.201	0.011	1.138	26.316x
16	64	16	1.662	0.028	1.599	19.021x
16	128	1	0.393	0.002	0.329	80.506x
16	128	2	0.540	0.003	0.476	58.560x
16	128	4	0.840	0.008	0.776	37.637x
16	128	8	1.203	0.021	1.140	26.270x
16	128	16	1.636	0.025	1.572	19.324x

Continued on next page

Table A.8: Results of benchmarking illust-vr using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	256	1	0.321	0.005	0.257	98.561x
16	256	2	0.513	0.007	0.450	61.595x
16	256	4	0.864	0.013	0.801	36.584x
16	256	8	1.109	0.018	1.046	28.500x
16	256	16	1.213	0.014	1.150	26.058x
32	16	1	1.075	0.002	1.012	29.396x
32	16	2	1.154	0.000	1.091	27.388x
32	16	4	1.294	0.001	1.230	24.440x
32	16	8	1.363	0.001	1.300	23.188x
32	16	16	1.502	0.003	1.438	21.048x
32	32	1	0.596	0.001	0.532	53.072x
32	32	2	0.650	0.001	0.586	48.664x
32	32	4	0.787	0.001	0.724	40.151x
32	32	8	0.882	0.002	0.818	35.856x
32	32	16	1.103	0.033	1.040	28.660x
32	64	1	0.328	0.001	0.264	96.472x
32	64	2	0.382	0.001	0.318	82.858x
32	64	4	0.508	0.003	0.445	62.186x
32	64	8	0.715	0.045	0.651	44.234x
32	64	16	1.286	0.045	1.223	24.583x
32	128	1	0.230	0.003	0.167	137.254x
32	128	2	0.323	0.007	0.260	97.819x
32	128	4	0.539	0.013	0.475	58.690x
32	128	8	1.042	0.036	0.978	30.346x
32	128	16	1.500	0.029	1.437	21.075x
32	256	1	0.223	0.005	0.160	141.714x
32	256	2	0.377	0.013	0.313	83.949x
32	256	4	0.714	0.020	0.650	44.285x
32	256	8	0.971	0.013	0.907	32.563x
32	256	16	1.106	0.019	1.042	28.585x
64	16	1	0.542	0.002	0.479	58.290x
64	16	2	0.584	0.001	0.521	54.091x
64	16	4	0.654	0.001	0.591	48.316x
64	16	8	0.692	0.001	0.629	45.664x
64	16	16	0.778	0.004	0.714	40.645x
64	32	1	0.311	0.001	0.248	101.591x

Continued on next page

Table A.8: Results of benchmarking illust-vr using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	32	2	0.337	0.001	0.274	93.768x
64	32	4	0.408	0.002	0.345	77.449x
64	32	8	0.480	0.006	0.416	65.912x
64	32	16	0.777	0.043	0.713	40.696x
64	64	1	0.200	0.003	0.137	158.054x
64	64	2	0.238	0.008	0.174	132.863x
64	64	4	0.329	0.021	0.265	96.219x
64	64	8	0.609	0.028	0.546	51.889x
64	64	16	1.263	0.069	1.200	25.027x
64	128	1	0.181	0.004	0.117	174.885x
64	128	2	0.266	0.016	0.203	118.755x
64	128	4	0.500	0.019	0.436	63.253x
64	128	8	1.033	0.030	0.969	30.604x
64	128	16	1.477	0.032	1.414	21.403x
64	256	1	0.200	0.005	0.136	158.119x
64	256	2	0.362	0.013	0.298	87.440x
64	256	4	0.695	0.018	0.632	45.480x
64	256	8	0.966	0.016	0.903	32.723x
64	256	16	1.103	0.021	1.040	28.661x
128	16	1	0.295	0.003	0.231	107.222x
128	16	2	0.324	0.002	0.261	97.550x
128	16	4	0.376	0.002	0.312	84.174x
128	16	8	0.426	0.006	0.362	74.246x
128	16	16	0.628	0.012	0.565	50.335x
128	32	1	0.196	0.002	0.133	161.197x
128	32	2	0.230	0.003	0.166	137.523x
128	32	4	0.302	0.008	0.239	104.682x
128	32	8	0.502	0.022	0.439	62.975x
128	32	16	1.068	0.078	1.005	29.597x
128	64	1	0.176	0.004	0.113	179.572x
128	64	2	0.236	0.006	0.172	134.011x
128	64	4	0.392	0.014	0.328	80.686x
128	64	8	0.826	0.042	0.762	38.277x
128	64	16	1.390	0.034	1.327	22.739x
128	128	1	0.186	0.005	0.123	169.581x
128	128	2	0.303	0.013	0.240	104.231x

Continued on next page

Table A.8: Results of benchmarking illust-vr using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	128	4	0.587	0.020	0.523	53.899x
128	128	8	0.974	0.026	0.910	32.463x
128	128	16	1.206	0.026	1.142	26.217x
128	256	1	0.199	0.012	0.135	159.006x
128	256	2	0.358	0.014	0.294	88.381x
128	256	4	0.675	0.016	0.611	46.848x
128	256	8	0.932	0.015	0.869	33.912x
128	256	16	1.022	0.018	0.959	30.931x
256	16	1	0.205	0.003	0.142	154.046x
256	16	2	0.245	0.009	0.182	128.960x
256	16	4	0.306	0.007	0.242	103.474x
256	16	8	0.499	0.047	0.435	63.373x
256	16	16	1.022	0.034	0.959	30.930x
256	**32**	**1**	**0.174**	**0.005**	**0.111**	**181.670x**
256	32	2	0.233	0.006	0.170	135.643x
256	32	4	0.412	0.036	0.349	76.729x
256	32	8	0.880	0.029	0.816	35.938x
256	32	16	1.499	0.036	1.435	21.094x
256	64	1	0.185	0.011	0.122	170.842x
256	64	2	0.304	0.022	0.241	103.852x
256	64	4	0.587	0.024	0.523	53.897x
256	64	8	1.004	0.027	0.941	31.488x
256	64	16	1.261	0.024	1.197	25.077x
256	128	1	0.196	0.011	0.133	161.064x
256	128	2	0.343	0.013	0.280	92.095x
256	128	4	0.645	0.023	0.582	49.007x
256	128	8	0.994	0.020	0.931	31.803x
256	128	16	1.083	0.023	1.019	29.196x
512	16	1	0.194	0.004	0.131	162.596x
512	16	2	0.252	0.009	0.189	125.453x
512	16	4	0.372	0.014	0.308	85.075x
512	16	8	0.728	0.032	0.664	43.431x
512	16	16	1.328	0.054	1.264	23.806x
512	32	1	0.183	0.004	0.120	172.743x
512	32	2	0.295	0.011	0.232	107.045x
512	32	4	0.570	0.028	0.506	55.503x

Continued on next page

Table A.8: Results of benchmarking illust-vr using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	32	8	0.983	0.020	0.920	32.145x
512	32	16	1.273	0.019	1.210	24.830x
512	64	1	0.193	0.010	0.130	163.484x
512	64	2	0.332	0.016	0.268	95.339x
512	64	4	0.619	0.020	0.556	51.051x
512	64	8	0.992	0.023	0.928	31.880x
512	64	16	1.103	0.019	1.039	28.670x
1024	16	1	0.199	0.005	0.136	158.622x
1024	16	2	0.266	0.008	0.202	118.974x
1024	16	4	0.412	0.015	0.349	76.731x
1024	16	8	0.805	0.036	0.741	39.278x
1024	16	16	1.330	0.050	1.266	23.775x
1024	32	1	0.194	0.012	0.131	162.841x
1024	32	2	0.330	0.015	0.266	95.872x
1024	32	4	0.603	0.018	0.540	52.406x
1024	32	8	0.966	0.017	0.902	32.739x
1024	32	16	1.121	0.024	1.058	28.199x

A.2 Lic2d

Table A.9: Results of benchmarking lic2d using execution method "sequential"

mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
1.413	**0.013**	**1.408**	**1.000x**

Table A.10: Results of benchmarking lic2d using execution method "cuda"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.113	0.002	0.108	12.489x
16	32	0.067	0.000	0.062	21.133x
16	64	0.043	0.001	0.038	33.113x
16	128	0.030	0.000	0.025	47.122x
16	256	0.025	0.000	0.020	57.372x
16	512	0.023	0.000	0.019	60.764x
32	16	0.057	0.001	0.052	24.711x
32	32	0.033	0.000	0.028	42.592x
32	64	0.019	0.001	0.015	72.609x
32	128	0.014	0.000	0.010	97.461x
32	256	0.012	0.000	0.007	119.434x
32	512	0.011	0.000	0.006	131.899x
64	16	0.031	0.000	0.026	45.410x
64	32	0.018	0.000	0.013	79.058x
64	64	0.011	0.000	0.006	131.436x
64	128	0.008	0.000	0.003	186.997x
64	256	0.006	0.000	0.001	234.108x
64	512	0.006	0.000	0.001	251.563x
64	1024	0.005	0.000	0.001	263.446x
128	16	0.017	0.000	0.012	83.504x
128	32	0.011	0.000	0.006	126.305x
128	64	0.008	0.000	0.003	180.121x
128	128	0.006	0.000	0.001	232.620x
128	256	0.006	0.000	0.001	254.503x
128	512	0.005	0.000	0.001	263.552x
256	16	0.011	0.000	0.006	133.878x
256	32	0.008	0.000	0.004	167.552x
256	64	0.006	0.000	0.002	222.003x
256	128	0.006	0.000	0.001	250.113x

Continued on next page

Table A.10: Results of benchmarking lic2d using execution method "cuda" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	256	0.005	0.000	0.001	263.097x
512	16	0.007	0.000	0.003	190.210x
512	32	0.006	0.000	0.001	228.064x
512	64	0.005	0.000	0.001	266.349x
512	128	0.005	0.000	0.001	263.093x
1024	16	0.006	0.000	0.001	231.993x
1024	**32**	**0.005**	**0.000**	**0.000**	**271.204x**
1024	64	0.005	0.000	0.001	263.568x

Table A.11: Results of benchmarking lic2d using execution method "cuda-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.098	0.001	0.094	14.350x
16	32	0.056	0.002	0.051	25.126x
16	64	0.035	0.000	0.030	40.446x
16	128	0.032	0.001	0.027	44.784x
16	256	0.028	0.000	0.023	50.244x
16	512	0.029	0.000	0.024	48.496x
32	16	0.054	0.001	0.050	25.958x
32	32	0.030	0.000	0.025	47.398x
32	64	0.019	0.000	0.014	73.705x
32	128	0.016	0.000	0.011	87.391x
32	256	0.014	0.000	0.009	99.575x
32	512	0.015	0.000	0.010	96.145x
64	16	0.028	0.000	0.023	51.359x
64	32	0.015	0.000	0.011	92.646x
64	64	0.010	0.000	0.005	141.858x
64	128	0.008	0.000	0.004	167.328x
64	256	0.008	0.000	0.004	169.727x
64	512	0.010	0.000	0.005	146.372x
128	16	0.015	0.000	0.010	95.813x
128	32	0.010	0.000	0.005	142.775x
128	64	0.008	0.000	0.004	171.145x
128	128	0.008	0.000	0.003	183.571x
128	256	0.008	0.000	0.004	169.430x
256	16	0.009	0.000	0.004	153.507x

Continued on next page

Table A.11: Results of benchmarking lic2d using execution method "cuda-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	0.008	0.000	0.003	173.976x
256	64	0.007	0.000	0.003	188.918x
256	128	0.008	0.000	0.003	177.011x
512	16	0.007	0.000	0.002	206.793x
512	32	0.007	0.000	0.002	209.871x
512	64	0.007	0.000	0.003	193.344x
1024	**16**	**0.006**	**0.000**	**0.002**	**220.497x**
1024	32	0.007	0.000	0.002	203.502x

Table A.12: Results of benchmarking lic2d using execution method "cuda-batch"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.094	0.001	0.089	15.021x
16	32	0.053	0.000	0.048	26.798x
16	64	0.035	0.001	0.030	40.380x
16	128	0.026	0.000	0.021	54.022x
16	256	0.023	0.000	0.018	61.450x
16	512	0.023	0.000	0.018	62.612x
32	16	0.052	0.000	0.047	27.210x
32	32	0.028	0.000	0.024	49.636x
32	64	0.017	0.000	0.012	82.170x
32	128	0.013	0.000	0.008	112.977x
32	256	0.011	0.000	0.006	128.590x
32	512	0.010	0.000	0.005	138.880x
64	16	0.025	0.000	0.020	56.047x
64	32	0.014	0.000	0.009	101.637x
64	64	0.009	0.000	0.004	165.027x
64	128	0.006	0.000	0.002	222.903x
64	256	0.005	0.000	0.001	264.551x
64	512	0.005	0.000	0.000	279.983x
128	16	0.014	0.000	0.009	104.390x
128	32	0.009	0.000	0.004	156.803x
128	64	0.006	0.000	0.002	218.322x
128	128	0.005	0.000	0.001	261.613x
128	256	0.005	0.000	0.000	282.882x
256	16	0.009	0.000	0.004	161.400x

Continued on next page

Table A.12: Results of benchmarking lic2d using execution method "cuda-batch" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	0.007	0.000	0.002	195.566x
256	64	0.006	0.000	0.001	248.454x
256	128	0.005	0.000	0.000	272.464x
512	16	0.006	0.000	0.002	217.724x
512	32	0.005	0.000	0.001	259.925x
512	64	0.005	0.000	0.000	292.344x
1024	16	0.006	0.000	0.001	254.713x
1024	**32**	**0.005**	**0.000**	**0.000**	**298.662x**

Table A.13: Results of benchmarking lic2d using execution method "cuda-batch-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.099	0.002	0.094	14.323x
16	32	0.055	0.001	0.050	25.612x
16	64	0.035	0.000	0.030	40.429x
16	128	0.029	0.001	0.024	48.606x
16	256	0.028	0.000	0.023	50.261x
16	512	0.029	0.000	0.024	48.535x
32	16	0.055	0.000	0.051	25.491x
32	32	0.031	0.000	0.026	46.155x
32	64	0.019	0.000	0.015	72.851x
32	128	0.016	0.000	0.012	87.045x
32	256	0.014	0.000	0.009	99.580x
32	512	0.015	0.000	0.010	96.134x
64	16	0.028	0.000	0.023	51.332x
64	32	0.015	0.000	0.011	92.616x
64	64	0.010	0.000	0.005	141.921x
64	128	0.008	0.000	0.004	167.329x
64	256	0.008	0.000	0.004	170.171x
64	512	0.010	0.000	0.005	146.559x
128	16	0.015	0.000	0.010	95.866x
128	32	0.010	0.000	0.005	142.912x
128	64	0.008	0.000	0.004	170.841x
128	128	0.008	0.000	0.003	183.556x
128	256	0.008	0.000	0.004	169.718x
256	16	0.009	0.000	0.004	153.474x

Continued on next page

Table A.13: Results of benchmarking lic2d using execution method "cuda-batch-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	0.008	0.000	0.003	173.832x
256	64	0.007	0.000	0.003	189.128x
256	128	0.008	0.000	0.003	177.211x
512	16	0.007	0.000	0.002	206.672x
512	32	0.007	0.000	0.002	210.138x
512	64	0.007	0.000	0.003	193.592x
1024	**16**	**0.006**	**0.000**	**0.002**	**220.678x**
1024	32	0.007	0.000	0.002	203.611x

Table A.14: Results of benchmarking lic2d using execution method "cuda-unified-memory"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.170	0.002	0.165	8.328x
16	32	0.098	0.001	0.093	14.464x
16	64	0.072	0.001	0.068	19.554x
16	128	0.059	0.001	0.055	23.762x
16	256	0.056	0.001	0.051	25.214x
16	512	0.054	0.001	0.049	26.182x
32	16	0.121	0.002	0.116	11.683x
32	32	0.066	0.001	0.061	21.397x
32	64	0.054	0.002	0.049	26.281x
32	128	0.047	0.001	0.042	29.959x
32	256	0.046	0.001	0.042	30.473x
32	512	0.045	0.001	0.041	31.186x
64	16	0.095	0.001	0.090	14.890x
64	32	0.050	0.001	0.045	28.163x
64	64	0.048	0.001	0.043	29.445x
64	128	0.041	0.001	0.036	34.370x
64	**256**	**0.041**	**0.001**	**0.036**	**34.510x**
64	512	0.042	0.001	0.037	33.745x
128	16	0.083	0.002	0.078	16.977x
128	32	0.045	0.001	0.041	31.172x
128	64	0.045	0.001	0.040	31.487x
128	128	0.043	0.001	0.038	32.944x
128	256	0.043	0.001	0.038	33.077x

Continued on next page

Table A.14: Results of benchmarking lic2d using execution method "cuda-unified-memory" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	16	0.078	0.002	0.074	18.016x
256	32	0.048	0.001	0.043	29.421x
256	64	0.047	0.001	0.042	30.072x
256	128	0.045	0.001	0.041	31.156x
512	16	0.084	0.002	0.080	16.730x
512	32	0.052	0.001	0.047	27.106x
512	64	0.049	0.001	0.045	28.662x
1024	16	0.086	0.002	0.082	16.354x
1024	32	0.050	0.001	0.045	28.484x

Table A.15: Results of benchmarking lic2d using execution method "cuda-global-queue"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.091	0.001	0.086	15.524x
16	16	2	0.090	0.000	0.086	15.634x
16	16	4	0.090	0.000	0.085	15.735x
16	16	8	0.090	0.000	0.085	15.756x
16	16	16	0.090	0.000	0.086	15.635x
16	32	1	0.048	0.000	0.044	29.233x
16	32	2	0.047	0.000	0.042	29.980x
16	32	4	0.047	0.000	0.042	30.019x
16	32	8	0.049	0.002	0.044	28.863x
16	32	16	0.051	0.001	0.046	27.846x
16	64	1	0.028	0.001	0.023	51.077x
16	64	2	0.029	0.000	0.025	48.321x
16	64	4	0.029	0.000	0.025	48.032x
16	64	8	0.030	0.000	0.026	46.474x
16	64	16	0.033	0.000	0.028	42.860x
16	128	1	0.017	0.000	0.012	84.141x
16	128	2	0.019	0.000	0.014	75.704x
16	128	4	0.020	0.000	0.015	72.201x
16	128	8	0.021	0.000	0.017	66.290x
16	128	16	0.025	0.000	0.020	57.201x
16	256	1	0.012	0.000	0.007	122.339x
16	256	2	0.014	0.000	0.009	100.350x

Continued on next page

Table A.15: Results of benchmarking lic2d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	256	4	0.015	0.000	0.011	91.198x
16	256	8	0.018	0.000	0.013	77.687x
16	256	16	0.022	0.000	0.017	63.727x
16	512	1	0.010	0.000	0.005	145.368x
16	512	2	0.013	0.000	0.008	109.666x
16	512	4	0.015	0.000	0.010	95.372x
16	512	8	0.018	0.000	0.013	79.927x
16	512	16	0.022	0.000	0.018	63.213x
32	16	1	0.050	0.000	0.045	28.304x
32	16	2	0.050	0.000	0.045	28.419x
32	16	4	0.050	0.000	0.045	28.485x
32	16	8	0.049	0.000	0.045	28.559x
32	16	16	0.050	0.001	0.045	28.389x
32	32	1	0.027	0.000	0.023	51.859x
32	32	2	0.027	0.000	0.022	52.705x
32	32	4	0.027	0.000	0.022	52.867x
32	32	8	0.027	0.000	0.022	52.517x
32	32	16	0.028	0.000	0.023	50.615x
32	64	1	0.015	0.000	0.011	92.021x
32	64	2	0.016	0.000	0.011	89.438x
32	64	4	0.016	0.000	0.011	88.470x
32	64	8	0.016	0.000	0.012	86.145x
32	64	16	0.018	0.000	0.013	79.114x
32	128	1	0.010	0.000	0.005	146.300x
32	128	2	0.011	0.000	0.006	131.898x
32	128	4	0.011	0.000	0.006	127.056x
32	128	8	0.012	0.000	0.007	116.915x
32	128	16	0.014	0.001	0.009	100.999x
32	256	1	0.007	0.000	0.002	196.800x
32	256	2	0.008	0.000	0.004	167.539x
32	256	4	0.009	0.000	0.004	153.207x
32	256	8	0.011	0.000	0.006	133.806x
32	256	16	0.013	0.000	0.009	106.019x
32	512	1	0.006	0.000	0.002	223.843x
32	512	2	0.008	0.000	0.003	178.500x
32	512	4	0.009	0.001	0.004	157.956x

Continued on next page

Table A.15: Results of benchmarking lic2d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	512	8	0.010	0.000	0.006	135.253x
32	512	16	0.013	0.000	0.009	104.837x
64	16	1	0.026	0.000	0.021	53.979x
64	16	2	0.026	0.001	0.022	53.413x
64	16	4	0.026	0.000	0.022	53.616x
64	16	8	0.026	0.000	0.022	53.464x
64	16	16	0.027	0.000	0.022	52.810x
64	32	1	0.015	0.000	0.011	92.726x
64	32	2	0.015	0.000	0.010	94.486x
64	32	4	0.015	0.000	0.010	94.533x
64	32	8	0.015	0.000	0.010	93.595x
64	32	16	0.016	0.000	0.011	89.214x
64	64	1	0.009	0.000	0.005	151.341x
64	64	2	0.010	0.000	0.005	146.989x
64	64	4	0.010	0.000	0.005	146.654x
64	64	8	0.010	0.000	0.005	141.393x
64	64	16	0.011	0.000	0.006	133.432x
64	128	1	0.006	0.000	0.002	217.379x
64	128	2	0.007	0.000	0.002	200.840x
64	128	4	0.007	0.000	0.003	194.231x
64	128	8	0.008	0.000	0.003	180.866x
64	128	16	0.009	0.000	0.004	157.068x
64	256	1	0.005	0.000	0.001	263.775x
64	256	2	0.006	0.000	0.001	236.465x
64	256	4	0.006	0.000	0.002	221.534x
64	256	8	0.007	0.000	0.002	196.000x
64	256	16	0.009	0.000	0.005	152.784x
64	512	1	0.005	0.000	0.000	284.578x
64	512	2	0.006	0.000	0.001	246.664x
64	512	4	0.006	0.000	0.002	225.896x
64	512	8	0.008	0.000	0.003	187.079x
64	512	16	0.009	0.000	0.005	150.994x
128	16	1	0.015	0.000	0.010	97.351x
128	16	2	0.015	0.000	0.010	96.104x
128	16	4	0.015	0.000	0.010	96.076x
128	16	8	0.015	0.000	0.010	95.740x

Continued on next page

Table A.15: Results of benchmarking lic2d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	16	16	0.015	0.000	0.010	92.871x
128	32	1	0.009	0.001	0.005	152.066x
128	32	2	0.009	0.000	0.005	152.424x
128	32	4	0.009	0.000	0.005	151.508x
128	32	8	0.010	0.000	0.005	148.143x
128	32	16	0.010	0.000	0.006	136.790x
128	64	1	0.006	0.000	0.002	223.070x
128	64	2	0.007	0.000	0.002	208.566x
128	64	4	0.007	0.000	0.002	203.651x
128	64	8	0.007	0.000	0.003	189.860x
128	64	16	0.008	0.000	0.003	175.178x
128	128	1	0.005	0.000	0.000	273.568x
128	128	2	0.006	0.000	0.001	245.125x
128	128	4	0.006	0.000	0.001	232.973x
128	128	8	0.007	0.000	0.002	215.041x
128	128	16	0.007	0.000	0.003	188.640x
128	256	1	0.005	0.000	0.000	293.170x
128	256	2	0.005	0.000	0.001	260.773x
128	256	4	0.006	0.000	0.001	240.553x
128	256	8	0.007	0.000	0.002	215.285x
128	256	16	0.007	0.000	0.003	192.083x
256	16	1	0.009	0.000	0.004	156.965x
256	16	2	0.009	0.000	0.005	152.785x
256	16	4	0.009	0.000	0.005	152.641x
256	16	8	0.010	0.001	0.005	148.098x
256	16	16	0.010	0.000	0.005	141.145x
256	32	1	0.006	0.000	0.002	222.985x
256	32	2	0.007	0.000	0.002	210.753x
256	32	4	0.007	0.000	0.002	206.844x
256	32	8	0.007	0.000	0.003	192.384x
256	32	16	0.008	0.000	0.003	174.579x
256	64	1	0.005	0.000	0.001	269.391x
256	64	2	0.006	0.000	0.001	248.489x
256	64	4	0.006	0.000	0.001	236.548x
256	64	8	0.007	0.000	0.002	214.293x
256	64	16	0.008	0.000	0.003	188.050x

Continued on next page

Table A.15: Results of benchmarking lic2d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	**128**	**1**	**0.005**	**0.000**	**0.000**	**295.818x**
256	128	2	0.005	0.000	0.001	260.731x
256	128	4	0.006	0.000	0.001	242.055x
256	128	8	0.007	0.000	0.002	206.251x
256	128	16	0.008	0.000	0.003	179.313x
512	16	1	0.006	0.000	0.002	217.514x
512	16	2	0.007	0.000	0.002	203.112x
512	16	4	0.007	0.000	0.002	197.767x
512	16	8	0.007	0.000	0.003	189.820x
512	16	16	0.008	0.000	0.004	170.372x
512	32	1	0.005	0.000	0.000	270.989x
512	32	2	0.006	0.000	0.001	248.485x
512	32	4	0.006	0.000	0.001	238.857x
512	32	8	0.007	0.001	0.002	214.194x
512	32	16	0.007	0.000	0.003	193.278x
512	**64**	**1**	**0.005**	**0.000**	**0.000**	**295.443x**
512	64	2	0.005	0.000	0.001	264.105x
512	64	4	0.006	0.000	0.001	246.698x
512	64	8	0.007	0.000	0.002	207.747x
512	64	16	0.008	0.000	0.003	173.913x
1024	16	1	0.006	0.000	0.001	250.251x
1024	16	2	0.006	0.000	0.001	230.923x
1024	16	4	0.006	0.000	0.002	224.580x
1024	16	8	0.007	0.000	0.002	205.384x
1024	16	16	0.008	0.000	0.004	168.571x
1024	**32**	**1**	**0.005**	**0.000**	**0.000**	**295.473x**
1024	32	2	0.005	0.000	0.001	263.090x
1024	32	4	0.006	0.000	0.001	244.560x
1024	32	8	0.007	0.000	0.002	209.980x
1024	32	16	0.008	0.000	0.003	177.507x

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.100	0.001	0.095	14.132x
16	16	2	0.100	0.001	0.096	14.087x

Continued on next page

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	4	0.102	0.001	0.097	13.878x
16	16	8	0.102	0.001	0.097	13.888x
16	16	16	0.106	0.001	0.101	13.327x
16	32	1	0.053	0.000	0.049	26.428x
16	32	2	0.052	0.000	0.047	27.430x
16	32	4	0.052	0.000	0.047	27.188x
16	32	8	0.052	0.000	0.047	27.122x
16	32	16	0.056	0.001	0.051	25.245x
16	64	1	0.039	0.001	0.035	35.996x
16	64	2	0.038	0.001	0.034	36.756x
16	64	4	0.040	0.001	0.035	35.363x
16	64	8	0.040	0.001	0.035	35.167x
16	64	16	0.042	0.001	0.038	33.252x
16	128	1	0.039	0.001	0.034	36.518x
16	128	2	0.038	0.001	0.034	36.752x
16	128	4	0.039	0.001	0.034	36.190x
16	128	8	0.040	0.001	0.035	35.244x
16	128	16	0.043	0.001	0.038	32.905x
16	256	1	0.039	0.001	0.035	35.959x
16	256	2	0.040	0.001	0.035	35.278x
16	256	4	0.041	0.001	0.036	34.728x
16	256	8	0.043	0.001	0.038	33.209x
16	256	16	0.046	0.001	0.041	30.828x
16	512	1	0.039	0.001	0.034	36.653x
16	512	2	0.040	0.001	0.035	35.382x
16	512	4	0.041	0.001	0.036	34.299x
16	512	8	0.043	0.001	0.038	32.718x
16	512	16	0.048	0.001	0.043	29.481x
32	16	1	0.074	0.002	0.069	19.046x
32	16	2	0.073	0.001	0.068	19.354x
32	16	4	0.073	0.002	0.069	19.235x
32	16	8	0.074	0.002	0.069	19.160x
32	16	16	0.075	0.002	0.070	18.801x
32	32	1	0.040	0.001	0.036	34.985x
32	32	2	0.040	0.001	0.035	35.393x
32	32	4	0.040	0.001	0.035	35.287x

Continued on next page

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	32	8	0.040	0.001	0.035	35.409x
32	32	16	0.040	0.001	0.036	35.054x
32	64	1	0.039	0.001	0.035	35.886x
32	64	2	0.039	0.001	0.035	36.000x
32	64	4	0.039	0.001	0.035	35.985x
32	64	8	0.040	0.001	0.035	35.722x
32	64	16	0.040	0.001	0.035	35.158x
32	128	1	0.039	0.001	0.034	36.232x
32	128	2	0.039	0.001	0.034	36.231x
32	128	4	0.040	0.001	0.035	35.615x
32	128	8	0.040	0.001	0.036	34.984x
32	128	16	0.043	0.001	0.038	33.166x
32	256	1	0.039	0.001	0.035	35.770x
32	256	2	0.040	0.001	0.035	35.377x
32	256	4	0.041	0.001	0.036	34.560x
32	256	8	0.043	0.001	0.038	33.012x
32	256	16	0.046	0.001	0.041	30.865x
32	512	1	0.039	0.001	0.034	36.086x
32	512	2	0.041	0.001	0.036	34.736x
32	512	4	0.042	0.001	0.037	33.918x
32	512	8	0.043	0.001	0.039	32.504x
32	512	16	0.047	0.001	0.042	30.149x
64	16	1	0.073	0.001	0.068	19.408x
64	16	2	0.072	0.001	0.068	19.492x
64	16	4	0.073	0.001	0.068	19.472x
64	16	8	0.073	0.002	0.068	19.404x
64	16	16	0.073	0.001	0.068	19.381x
64	32	1	0.039	0.001	0.035	35.891x
64	32	2	0.040	0.001	0.035	35.436x
64	32	4	0.040	0.001	0.035	35.641x
64	32	8	0.040	0.001	0.035	35.443x
64	32	16	0.040	0.001	0.035	35.145x
64	64	1	0.039	0.001	0.034	36.196x
64	64	2	0.039	0.001	0.035	35.856x
64	64	4	0.039	0.001	0.035	36.012x
64	64	8	0.040	0.001	0.035	35.274x

Continued on next page

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	64	16	0.041	0.001	0.036	34.508x
64	128	1	0.039	0.001	0.034	36.077x
64	128	2	0.039	0.001	0.034	36.017x
64	128	4	0.040	0.001	0.035	35.445x
64	128	8	0.041	0.001	0.036	34.602x
64	128	16	0.043	0.001	0.038	33.074x
64	256	1	0.039	0.001	0.035	35.836x
64	256	2	0.040	0.001	0.036	35.028x
64	256	4	0.041	0.001	0.036	34.331x
64	256	8	0.043	0.001	0.038	33.147x
64	256	16	0.044	0.001	0.040	31.865x
64	512	1	0.039	0.001	0.034	36.408x
64	512	2	0.041	0.001	0.036	34.773x
64	512	4	0.041	0.001	0.036	34.325x
64	512	8	0.043	0.001	0.038	33.113x
64	512	16	0.044	0.001	0.039	32.353x
128	16	1	0.072	0.002	0.067	19.618x
128	16	2	0.073	0.002	0.068	19.482x
128	16	4	0.072	0.001	0.067	19.656x
128	16	8	0.072	0.002	0.068	19.496x
128	16	16	0.073	0.002	0.068	19.442x
128	**32**	**1**	**0.038**	**0.001**	**0.033**	**37.033x**
128	32	2	0.039	0.001	0.034	36.672x
128	32	4	0.039	0.001	0.034	36.563x
128	32	8	0.040	0.001	0.035	35.447x
128	32	16	0.040	0.001	0.035	35.180x
128	64	1	0.039	0.001	0.034	36.116x
128	64	2	0.039	0.001	0.035	35.864x
128	64	4	0.040	0.001	0.035	35.613x
128	64	8	0.041	0.001	0.036	34.853x
128	64	16	0.042	0.001	0.037	33.796x
128	128	1	0.039	0.001	0.034	36.273x
128	128	2	0.040	0.001	0.035	35.704x
128	128	4	0.040	0.001	0.036	35.115x
128	128	8	0.041	0.001	0.037	34.076x
128	128	16	0.044	0.001	0.039	32.386x

Continued on next page

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	256	1	0.039	0.001	0.034	36.064x
128	256	2	0.041	0.001	0.036	34.812x
128	256	4	0.041	0.001	0.036	34.292x
128	256	8	0.042	0.001	0.037	33.525x
128	256	16	0.045	0.001	0.040	31.686x
256	16	1	0.072	0.002	0.067	19.653x
256	16	2	0.072	0.002	0.067	19.591x
256	16	4	0.072	0.002	0.067	19.646x
256	16	8	0.072	0.002	0.068	19.555x
256	16	16	0.073	0.002	0.068	19.333x
256	32	1	0.039	0.001	0.034	36.149x
256	32	2	0.039	0.001	0.035	35.987x
256	32	4	0.040	0.001	0.035	35.575x
256	32	8	0.040	0.001	0.035	35.124x
256	32	16	0.042	0.001	0.037	33.475x
256	64	1	0.039	0.001	0.034	36.372x
256	64	2	0.040	0.001	0.035	35.713x
256	64	4	0.040	0.001	0.036	35.025x
256	64	8	0.042	0.001	0.037	33.532x
256	64	16	0.045	0.001	0.040	31.251x
256	128	1	0.039	0.001	0.034	36.345x
256	128	2	0.040	0.001	0.036	35.062x
256	128	4	0.041	0.001	0.037	34.205x
256	128	8	0.044	0.001	0.039	32.070x
256	128	16	0.047	0.001	0.042	30.319x
512	16	1	0.072	0.001	0.067	19.603x
512	16	2	0.073	0.002	0.068	19.473x
512	16	4	0.073	0.001	0.068	19.447x
512	16	8	0.073	0.001	0.068	19.302x
512	16	16	0.075	0.002	0.070	18.961x
512	32	1	0.039	0.001	0.034	36.094x
512	32	2	0.040	0.001	0.035	35.153x
512	32	4	0.041	0.001	0.036	34.396x
512	32	8	0.042	0.001	0.038	33.427x
512	32	16	0.046	0.001	0.042	30.553x
512	64	1	0.039	0.001	0.034	36.334x

Continued on next page

Table A.16: Results of benchmarking lic2d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	64	2	0.040	0.001	0.036	34.958x
512	64	4	0.041	0.001	0.037	34.123x
512	64	8	0.043	0.001	0.039	32.643x
512	64	16	0.051	0.001	0.047	27.463x
1024	16	1	0.073	0.002	0.068	19.433x
1024	16	2	0.074	0.001	0.069	19.153x
1024	16	4	0.074	0.002	0.069	19.147x
1024	16	8	0.075	0.002	0.070	18.846x
1024	16	16	0.077	0.002	0.072	18.402x
1024	32	1	0.039	0.001	0.034	36.397x
1024	32	2	0.040	0.001	0.036	35.091x
1024	32	4	0.041	0.001	0.037	34.051x
1024	32	8	0.044	0.001	0.039	32.379x
1024	32	16	0.052	0.001	0.048	26.980x

A.3 Mandelbrot

Table A.17: Results of benchmarking mandelbrot using execution method "sequential"

mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
6.419	**0.051**	**6.319**	**1.000x**

Table A.18: Results of benchmarking mandelbrot using execution method "cuda"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	21.277	0.005	21.178	0.302x
16	32	13.214	0.002	13.114	0.486x
16	64	7.774	0.002	7.675	0.826x
16	128	3.923	0.009	3.824	1.636x
16	256	2.985	0.002	2.886	2.150x
16	512	1.968	0.002	1.869	3.261x
16	1024	1.851	0.001	1.752	3.467x
32	16	10.564	0.001	10.464	0.608x
32	32	6.646	0.001	6.546	0.966x
32	64	3.244	0.002	3.144	1.979x
32	128	2.499	0.002	2.400	2.568x
32	256	1.574	0.001	1.474	4.079x
32	512	1.062	0.001	0.963	6.043x
32	1024	0.976	0.000	0.876	6.580x
64	16	5.432	0.001	5.333	1.182x
64	32	2.914	0.001	2.815	2.202x
64	64	2.059	0.001	1.959	3.118x
64	128	1.383	0.001	1.284	4.641x
64	256	0.881	0.001	0.782	7.283x
64	512	0.622	0.000	0.522	10.327x
64	**1024**	**0.302**	**0.001**	**0.202**	**21.284x**
128	16	2.475	0.001	2.376	2.593x
128	32	1.807	0.001	1.707	3.553x
128	64	1.176	0.001	1.077	5.458x
128	128	0.770	0.000	0.670	8.341x
128	256	0.506	0.001	0.406	12.690x
128	512	0.302	0.002	0.202	21.277x
256	16	1.440	0.000	1.341	4.456x

Continued on next page

Table A.18: Results of benchmarking mandelbrot using execution method "cuda" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	1.074	0.001	0.974	5.978x
256	64	0.691	0.000	0.592	9.287x
256	128	0.463	0.001	0.363	13.878x
256	256	0.303	0.001	0.204	21.184x
512	16	0.840	0.000	0.741	7.637x
512	32	0.653	0.001	0.554	9.829x
512	64	0.438	0.001	0.338	14.670x
512	128	0.303	0.001	0.204	21.184x
1024	16	0.536	0.003	0.436	11.978x
1024	32	0.427	0.002	0.327	15.044x
1024	64	0.304	0.001	0.204	21.147x

Table A.19: Results of benchmarking mandelbrot using execution method "cuda-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	9.527	0.002	9.428	0.674x
16	32	4.743	0.003	4.643	1.353x
16	64	2.596	0.004	2.497	2.472x
16	128	1.639	0.004	1.540	3.916x
16	256	1.248	0.002	1.149	5.142x
16	512	1.113	0.002	1.014	5.766x
16	1024	1.102	0.003	1.003	5.824x
32	16	4.797	0.002	4.697	1.338x
32	32	2.434	0.002	2.335	2.637x
32	64	1.289	0.005	1.190	4.978x
32	128	0.893	0.003	0.793	7.191x
32	256	0.717	0.002	0.618	8.948x
32	512	0.649	0.002	0.549	9.894x
32	1024	0.650	0.002	0.550	9.881x
64	16	2.408	0.002	2.309	2.665x
64	32	1.205	0.002	1.106	5.325x
64	64	0.640	0.004	0.540	10.035x
64	128	0.475	0.002	0.376	13.514x
64	256	0.408	0.002	0.308	15.738x
64	512	0.379	0.001	0.279	16.950x
128	16	1.209	0.003	1.109	5.310x

Continued on next page

Table A.19: Results of benchmarking mandelbrot using execution method "cuda-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	32	0.628	0.002	0.528	10.224x
128	64	0.368	0.002	0.269	17.439x
128	128	0.314	0.001	0.215	20.423x
128	256	0.306	0.001	0.206	20.986x
256	16	0.609	0.002	0.509	10.542x
256	32	0.347	0.001	0.247	18.523x
256	64	0.254	0.001	0.155	25.240x
256	128	0.228	0.001	0.128	28.212x
512	16	0.334	0.003	0.235	19.213x
512	32	0.219	0.001	0.120	29.277x
512	64	0.185	0.001	0.086	34.683x
1024	16	0.222	0.001	0.122	28.956x
1024	**32**	**0.180**	**0.001**	**0.080**	**35.755x**

Table A.20: Results of benchmarking mandelbrot using execution method "cuda-batch"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	9.405	0.001	9.306	0.682x
16	32	4.883	0.000	4.783	1.315x
16	64	2.513	0.002	2.414	2.554x
16	128	1.891	0.001	1.791	3.395x
16	256	1.629	0.001	1.529	3.941x
16	512	1.485	0.001	1.385	4.323x
16	1024	1.354	0.000	1.255	4.740x
32	16	4.676	0.001	4.576	1.373x
32	32	2.370	0.000	2.271	2.708x
32	64	1.260	0.001	1.161	5.093x
32	128	0.950	0.001	0.851	6.754x
32	256	0.829	0.000	0.730	7.742x
32	512	0.673	0.000	0.574	9.538x
32	1024	0.690	0.000	0.590	9.306x
64	16	2.361	0.000	2.261	2.719x
64	32	1.224	0.002	1.124	5.245x
64	64	0.650	0.001	0.550	9.879x
64	128	0.494	0.001	0.395	12.989x

Continued on next page

Table A.20: Results of benchmarking mandelbrot using execution method "cuda-batch" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	256	0.409	0.000	0.310	15.679x
64	512	0.379	0.000	0.280	16.916x
128	16	1.106	0.002	1.006	5.804x
128	32	0.617	0.000	0.518	10.395x
128	64	0.361	0.001	0.261	17.798x
128	128	0.281	0.000	0.181	22.858x
128	256	0.268	0.000	0.169	23.954x
256	16	0.618	0.002	0.518	10.390x
256	32	0.331	0.001	0.232	19.378x
256	64	0.232	0.001	0.132	27.679x
256	128	0.204	0.000	0.105	31.447x
512	16	0.322	0.003	0.223	19.905x
512	32	0.210	0.001	0.110	30.588x
512	64	0.187	0.000	0.088	34.286x
1024	16	0.224	0.003	0.125	28.624x
1024	**32**	**0.177**	**0.000**	**0.077**	**36.315x**

Table A.21: Results of benchmarking mandelbrot using execution method "cuda-batch-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	9.530	0.003	9.431	0.674x
16	32	4.742	0.003	4.642	1.354x
16	64	2.597	0.004	2.498	2.472x
16	128	1.640	0.004	1.541	3.914x
16	256	1.249	0.002	1.150	5.137x
16	512	1.114	0.002	1.015	5.761x
16	1024	1.101	0.003	1.002	5.830x
32	16	4.798	0.002	4.698	1.338x
32	32	2.435	0.002	2.335	2.636x
32	64	1.288	0.004	1.188	4.985x
32	128	0.892	0.003	0.792	7.199x
32	256	0.718	0.002	0.618	8.945x
32	512	0.650	0.001	0.551	9.873x
32	1024	0.649	0.002	0.550	9.890x
64	16	2.408	0.002	2.309	2.665x
64	32	1.205	0.002	1.106	5.327x

Continued on next page

Table A.21: Results of benchmarking mandelbrot using execution method "cuda-batch-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	64	0.641	0.003	0.541	10.021x
64	128	0.475	0.002	0.376	13.512x
64	256	0.408	0.001	0.308	15.740x
64	512	0.379	0.001	0.279	16.948x
128	16	1.209	0.003	1.110	5.307x
128	32	0.628	0.002	0.529	10.221x
128	64	0.368	0.002	0.269	17.423x
128	128	0.314	0.001	0.215	20.414x
128	256	0.306	0.001	0.207	20.977x
256	16	0.609	0.003	0.510	10.539x
256	32	0.346	0.002	0.247	18.550x
256	64	0.254	0.001	0.155	25.262x
256	128	0.228	0.001	0.128	28.190x
512	16	0.335	0.003	0.235	19.188x
512	32	0.219	0.001	0.120	29.277x
512	64	0.185	0.001	0.085	34.779x
1024	16	0.222	0.002	0.123	28.894x
1024	**32**	**0.179**	**0.001**	**0.080**	**35.801x**

Table A.22: Results of benchmarking mandelbrot using execution method "cuda-unified-memory"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	21.345	0.006	21.246	0.301x
16	32	13.294	0.007	13.195	0.483x
16	64	7.880	0.008	7.780	0.815x
16	128	4.016	0.007	3.917	1.598x
16	256	3.087	0.007	2.988	2.079x
16	512	2.030	0.017	1.931	3.161x
16	1024	1.937	0.005	1.838	3.313x
32	16	10.687	0.008	10.587	0.601x
32	32	6.743	0.010	6.644	0.952x
32	64	3.343	0.012	3.244	1.920x
32	128	2.594	0.009	2.494	2.475x
32	256	1.681	0.007	1.581	3.819x
32	512	1.124	0.003	1.025	5.709x

Continued on next page

Table A.22: Results of benchmarking mandelbrot using execution method "cuda-unified-memory" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	1024	1.018	0.002	0.919	6.304x
64	16	5.583	0.014	5.484	1.150x
64	32	3.058	0.022	2.959	2.099x
64	64	2.141	0.004	2.042	2.998x
64	128	1.441	0.024	1.342	4.454x
64	256	0.911	0.002	0.811	7.048x
64	512	0.648	0.001	0.549	9.901x
128	16	2.566	0.007	2.466	2.502x
128	32	1.864	0.019	1.764	3.444x
128	64	1.212	0.002	1.113	5.294x
128	128	0.802	0.002	0.702	8.007x
128	256	0.538	0.003	0.438	11.940x
256	16	1.488	0.006	1.389	4.313x
256	32	1.112	0.005	1.013	5.770x
256	64	0.734	0.009	0.635	8.745x
256	128	0.510	0.002	0.411	12.582x
512	16	0.930	0.004	0.831	6.902x
512	32	0.704	0.011	0.604	9.122x
512	64	0.471	0.007	0.372	13.628x
1024	16	0.582	0.009	0.483	11.022x
1024	**32**	**0.470**	**0.004**	**0.370**	**13.671x**

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	1.822	0.003	1.723	3.523x
16	16	2	1.892	0.001	1.793	3.392x
16	16	4	2.110	0.000	2.011	3.042x
16	16	8	2.312	0.000	2.213	2.776x
16	16	16	2.669	0.001	2.570	2.405x
16	32	1	0.995	0.000	0.896	6.450x
16	32	2	1.099	0.000	1.000	5.839x
16	32	4	1.084	0.000	0.984	5.922x
16	32	8	1.289	0.000	1.190	4.978x
16	32	16	1.666	0.000	1.566	3.854x

Continued on next page

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	64	1	0.525	0.000	0.426	12.226x
16	64	2	0.623	0.000	0.523	10.305x
16	64	4	0.730	0.001	0.631	8.788x
16	64	8	0.803	0.001	0.704	7.992x
16	64	16	0.970	0.001	0.871	6.618x
16	128	1	0.387	0.000	0.287	16.606x
16	128	2	0.430	0.000	0.330	14.944x
16	128	4	0.595	0.001	0.495	10.796x
16	128	8	0.615	0.001	0.515	10.438x
16	128	16	0.667	0.002	0.567	9.625x
16	256	1	0.286	0.001	0.186	22.458x
16	256	2	0.312	0.000	0.213	20.564x
16	256	4	0.535	0.001	0.436	11.996x
16	256	8	0.542	0.001	0.443	11.835x
16	256	16	0.558	0.003	0.459	11.500x
16	512	1	0.220	0.000	0.120	29.194x
16	512	2	0.289	0.001	0.189	22.216x
16	512	4	0.509	0.001	0.409	12.623x
16	512	8	0.513	0.002	0.413	12.521x
16	512	16	0.525	0.004	0.425	12.238x
16	1024	1	0.207	0.000	0.108	30.972x
16	1024	2	0.284	0.000	0.185	22.600x
16	1024	4	0.519	0.001	0.419	12.370x
16	1024	8	0.520	0.004	0.421	12.340x
16	1024	16	0.537	0.007	0.438	11.946x
32	16	1	0.907	0.000	0.808	7.076x
32	16	2	0.942	0.000	0.843	6.812x
32	16	4	1.052	0.000	0.953	6.099x
32	16	8	1.152	0.001	1.052	5.574x
32	16	16	1.330	0.003	1.231	4.825x
32	32	1	0.497	0.000	0.397	12.917x
32	32	2	0.549	0.000	0.449	11.700x
32	32	4	0.543	0.000	0.444	11.821x
32	32	8	0.646	0.000	0.546	9.938x
32	32	16	0.834	0.001	0.735	7.695x
32	64	1	0.265	0.000	0.166	24.221x

Continued on next page

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	64	2	0.312	0.000	0.212	20.585x
32	64	4	0.366	0.000	0.267	17.514x
32	64	8	0.404	0.001	0.304	15.901x
32	64	16	0.488	0.001	0.389	13.151x
32	128	1	0.197	0.000	0.098	32.584x
32	128	2	0.218	0.000	0.119	29.377x
32	128	4	0.300	0.001	0.200	21.413x
32	128	8	0.311	0.001	0.212	20.623x
32	128	16	0.340	0.002	0.241	18.877x
32	256	1	0.150	0.000	0.050	42.865x
32	256	2	0.158	0.000	0.059	40.631x
32	256	4	0.270	0.001	0.171	23.738x
32	256	8	0.276	0.001	0.177	23.252x
32	256	16	0.288	0.003	0.189	22.263x
32	512	1	0.117	0.000	0.017	54.971x
32	512	2	0.147	0.000	0.047	43.749x
32	512	4	0.258	0.001	0.159	24.875x
32	512	8	0.265	0.002	0.166	24.191x
32	512	16	0.281	0.005	0.182	22.836x
32	1024	1	0.111	0.000	0.012	57.639x
32	1024	2	0.144	0.000	0.045	44.488x
32	1024	4	0.266	0.002	0.167	24.103x
32	1024	8	0.280	0.006	0.180	22.938x
32	1024	16	0.295	0.009	0.195	21.765x
64	16	1	0.456	0.000	0.357	14.067x
64	16	2	0.475	0.000	0.375	13.523x
64	16	4	0.531	0.000	0.431	12.096x
64	16	8	0.581	0.000	0.482	11.041x
64	16	16	0.673	0.000	0.573	9.543x
64	32	1	0.254	0.000	0.155	25.235x
64	32	2	0.288	0.000	0.189	22.285x
64	32	4	0.278	0.000	0.179	23.077x
64	32	8	0.332	0.001	0.233	19.331x
64	32	16	0.429	0.000	0.329	14.970x
64	64	1	0.155	0.000	0.056	41.421x
64	64	2	0.182	0.001	0.082	35.350x

Continued on next page

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	64	4	0.199	0.001	0.099	32.324x
64	64	8	0.220	0.001	0.121	29.139x
64	64	16	0.264	0.001	0.165	24.295x
64	128	1	0.129	0.000	0.030	49.730x
64	128	2	0.142	0.000	0.043	45.074x
64	128	4	0.178	0.001	0.079	35.998x
64	128	8	0.186	0.001	0.086	34.550x
64	128	16	0.204	0.002	0.104	31.529x
64	256	1	0.110	0.000	0.010	58.440x
64	256	2	0.120	0.000	0.020	53.563x
64	256	4	0.166	0.001	0.067	38.555x
64	256	8	0.172	0.002	0.073	37.294x
64	256	16	0.187	0.004	0.088	34.247x
64	512	1	0.100	0.000	0.000	64.312x
64	512	2	0.114	0.000	0.015	56.083x
64	512	4	0.167	0.001	0.068	38.437x
64	512	8	0.173	0.002	0.073	37.169x
64	512	16	0.197	0.004	0.098	32.517x
128	16	1	0.236	0.003	0.137	27.159x
128	16	2	0.247	0.000	0.148	25.986x
128	16	4	0.298	0.000	0.199	21.511x
128	16	8	0.320	0.000	0.221	20.046x
128	16	16	0.363	0.000	0.264	17.684x
128	32	1	0.151	0.000	0.052	42.494x
128	32	2	0.172	0.000	0.073	37.334x
128	32	4	0.185	0.000	0.086	34.634x
128	32	8	0.208	0.000	0.109	30.839x
128	32	16	0.253	0.001	0.154	25.336x
128	64	1	0.121	0.001	0.021	53.141x
128	64	2	0.133	0.000	0.034	48.087x
128	64	4	0.160	0.000	0.060	40.139x
128	64	8	0.169	0.001	0.069	38.077x
128	64	16	0.189	0.002	0.090	33.913x
128	128	1	0.106	0.000	0.006	60.649x
128	128	2	0.118	0.000	0.019	54.246x
128	128	4	0.152	0.001	0.053	42.127x

Continued on next page

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	128	8	0.157	0.002	0.057	40.916x
128	128	16	0.175	0.003	0.076	36.587x
128	256	1	0.100	0.000	0.001	63.926x
128	256	2	0.115	0.000	0.015	55.909x
128	256	4	0.155	0.001	0.055	41.517x
128	256	8	0.163	0.003	0.063	39.475x
128	256	16	0.186	0.005	0.087	34.521x
256	16	1	0.140	0.001	0.041	45.726x
256	16	2	0.167	0.000	0.068	38.410x
256	16	4	0.200	0.000	0.101	32.029x
256	16	8	0.207	0.000	0.107	31.026x
256	16	16	0.229	0.001	0.130	27.993x
256	32	1	0.123	0.000	0.024	51.984x
256	32	2	0.131	0.000	0.031	49.074x
256	32	4	0.157	0.001	0.058	40.856x
256	32	8	0.165	0.001	0.066	38.854x
256	32	16	0.185	0.002	0.086	34.612x
256	64	1	0.105	0.000	0.006	61.063x
256	64	2	0.118	0.000	0.019	54.214x
256	64	4	0.151	0.001	0.052	42.483x
256	64	8	0.155	0.001	0.056	41.353x
256	64	16	0.172	0.004	0.072	37.363x
256	128	1	0.100	0.001	0.001	64.148x
256	128	2	0.113	0.000	0.014	56.643x
256	128	4	0.152	0.001	0.053	42.109x
256	128	8	0.163	0.003	0.064	39.274x
256	128	16	0.188	0.005	0.089	34.129x
512	16	1	0.111	0.002	0.012	57.812x
512	16	2	0.131	0.000	0.031	49.027x
512	16	4	0.167	0.000	0.068	38.366x
512	16	8	0.164	0.001	0.064	39.224x
512	16	16	0.180	0.001	0.080	35.738x
512	32	1	0.105	0.000	0.005	61.187x
512	32	2	0.119	0.001	0.019	54.053x
512	32	4	0.151	0.001	0.051	42.575x
512	32	8	0.154	0.001	0.055	41.580x

Continued on next page

Table A.23: Results of benchmarking mandelbrot using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	32	16	0.170	0.003	0.070	37.838x
512	64	1	0.100	0.000	0.001	64.160x
512	64	2	0.113	0.002	0.014	56.613x
512	64	4	0.151	0.001	0.052	42.485x
512	64	8	0.157	0.002	0.057	40.917x
512	64	16	0.176	0.004	0.076	36.538x
1024	**16**	**1**	**0.099**	**0.001**	**0.000**	**64.560x**
1024	16	2	0.122	0.000	0.023	52.638x
1024	16	4	0.157	0.000	0.057	41.005x
1024	16	8	0.154	0.001	0.055	41.637x
1024	16	16	0.170	0.002	0.071	37.661x
1024	32	1	0.100	0.000	0.001	64.114x
1024	32	2	0.113	0.000	0.014	56.838x
1024	32	4	0.150	0.001	0.051	42.723x
1024	32	8	0.154	0.002	0.054	41.722x
1024	32	16	0.173	0.003	0.073	37.179x

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	1.822	0.002	1.723	3.523x
16	16	2	1.893	0.000	1.793	3.392x
16	16	4	2.111	0.001	2.011	3.041x
16	16	8	2.312	0.000	2.213	2.776x
16	16	16	2.670	0.001	2.571	2.404x
16	32	1	0.996	0.000	0.896	6.448x
16	32	2	1.099	0.000	1.000	5.839x
16	32	4	1.084	0.000	0.985	5.920x
16	32	8	1.290	0.000	1.190	4.976x
16	32	16	1.666	0.001	1.567	3.853x
16	64	1	0.526	0.000	0.426	12.212x
16	64	2	0.623	0.000	0.524	10.299x
16	64	4	0.731	0.001	0.632	8.782x
16	64	8	0.804	0.001	0.704	7.985x
16	64	16	0.970	0.001	0.871	6.615x

Continued on next page

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	128	1	0.387	0.000	0.288	16.588x
16	128	2	0.430	0.000	0.331	14.929x
16	128	4	0.595	0.001	0.496	10.785x
16	128	8	0.615	0.001	0.516	10.435x
16	128	16	0.667	0.002	0.568	9.618x
16	256	1	0.286	0.000	0.187	22.428x
16	256	2	0.313	0.000	0.213	20.525x
16	256	4	0.536	0.001	0.436	11.983x
16	256	8	0.543	0.001	0.443	11.824x
16	256	16	0.558	0.003	0.459	11.495x
16	512	1	0.220	0.000	0.121	29.129x
16	512	2	0.289	0.000	0.190	22.177x
16	512	4	0.509	0.001	0.409	12.616x
16	512	8	0.514	0.003	0.414	12.497x
16	512	16	0.526	0.005	0.427	12.200x
16	1024	1	0.208	0.000	0.108	30.909x
16	1024	2	0.285	0.000	0.185	22.558x
16	1024	4	0.519	0.001	0.420	12.363x
16	1024	8	0.520	0.003	0.421	12.336x
16	1024	16	0.541	0.008	0.442	11.859x
32	16	1	0.908	0.000	0.808	7.073x
32	16	2	0.943	0.000	0.843	6.808x
32	16	4	1.053	0.000	0.954	6.096x
32	16	8	1.153	0.001	1.054	5.565x
32	16	16	1.333	0.001	1.233	4.816x
32	32	1	0.497	0.000	0.398	12.906x
32	32	2	0.549	0.000	0.450	11.687x
32	32	4	0.544	0.000	0.444	11.808x
32	32	8	0.647	0.001	0.547	9.927x
32	32	16	0.835	0.000	0.736	7.685x
32	64	1	0.266	0.000	0.167	24.127x
32	64	2	0.313	0.000	0.214	20.505x
32	64	4	0.368	0.000	0.269	17.441x
32	64	8	0.405	0.001	0.306	15.841x
32	64	16	0.489	0.001	0.390	13.114x
32	128	1	0.198	0.000	0.099	32.412x

Continued on next page

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	128	2	0.220	0.000	0.120	29.231x
32	128	4	0.301	0.001	0.201	21.353x
32	128	8	0.312	0.001	0.212	20.597x
32	128	16	0.341	0.002	0.241	18.846x
32	256	1	0.150	0.000	0.051	42.705x
32	256	2	0.158	0.000	0.059	40.511x
32	256	4	0.271	0.001	0.171	23.695x
32	256	8	0.276	0.001	0.177	23.237x
32	256	16	0.289	0.003	0.190	22.191x
32	512	1	0.117	0.000	0.018	54.750x
32	512	2	0.147	0.000	0.048	43.661x
32	512	4	0.258	0.001	0.159	24.833x
32	512	8	0.266	0.002	0.166	24.154x
32	512	16	0.283	0.005	0.184	22.671x
32	1024	1	0.112	0.000	0.012	57.393x
32	1024	2	0.145	0.000	0.045	44.336x
32	1024	4	0.267	0.002	0.168	24.039x
32	1024	8	0.279	0.006	0.179	23.048x
32	1024	16	0.294	0.008	0.195	21.815x
64	16	1	0.457	0.000	0.357	14.053x
64	16	2	0.475	0.000	0.376	13.507x
64	16	4	0.531	0.000	0.432	12.084x
64	16	8	0.582	0.001	0.483	11.028x
64	16	16	0.673	0.000	0.574	9.531x
64	32	1	0.255	0.000	0.155	25.182x
64	32	2	0.289	0.000	0.189	22.246x
64	32	4	0.280	0.002	0.180	22.963x
64	32	8	0.333	0.001	0.233	19.285x
64	32	16	0.429	0.000	0.330	14.951x
64	64	1	0.155	0.000	0.056	41.283x
64	64	2	0.182	0.000	0.083	35.281x
64	64	4	0.199	0.000	0.100	32.250x
64	64	8	0.221	0.001	0.121	29.079x
64	64	16	0.264	0.001	0.165	24.278x
64	128	1	0.130	0.000	0.030	49.562x
64	128	2	0.143	0.000	0.043	44.927x

Continued on next page

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	128	4	0.179	0.001	0.080	35.866x
64	128	8	0.186	0.001	0.087	34.427x
64	128	16	0.204	0.002	0.105	31.403x
64	256	1	0.110	0.000	0.011	58.185x
64	256	2	0.120	0.000	0.021	53.286x
64	256	4	0.167	0.001	0.068	38.428x
64	256	8	0.173	0.001	0.073	37.166x
64	256	16	0.188	0.004	0.088	34.181x
64	512	1	0.100	0.000	0.001	64.001x
64	512	2	0.115	0.000	0.015	55.879x
64	512	4	0.167	0.001	0.068	38.373x
64	512	8	0.174	0.003	0.074	36.973x
64	512	16	0.198	0.006	0.099	32.346x
128	16	1	0.237	0.002	0.137	27.125x
128	16	2	0.248	0.001	0.149	25.885x
128	16	4	0.299	0.000	0.200	21.434x
128	16	8	0.321	0.000	0.222	19.984x
128	16	16	0.364	0.000	0.265	17.622x
128	32	1	0.152	0.000	0.052	42.270x
128	32	2	0.173	0.000	0.073	37.145x
128	32	4	0.186	0.000	0.087	34.455x
128	32	8	0.209	0.001	0.110	30.690x
128	32	16	0.254	0.001	0.155	25.252x
128	64	1	0.121	0.000	0.021	53.228x
128	64	2	0.134	0.000	0.035	47.927x
128	64	4	0.160	0.001	0.060	40.180x
128	64	8	0.169	0.001	0.070	37.984x
128	64	16	0.190	0.002	0.090	33.842x
128	128	1	0.106	0.000	0.007	60.282x
128	128	2	0.119	0.000	0.020	53.950x
128	128	4	0.153	0.001	0.054	41.868x
128	128	8	0.158	0.001	0.059	40.575x
128	128	16	0.175	0.003	0.076	36.612x
128	256	1	0.101	0.000	0.002	63.539x
128	256	2	0.115	0.001	0.016	55.666x
128	256	4	0.155	0.001	0.056	41.316x

Continued on next page

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	256	8	0.164	0.003	0.064	39.196x
128	256	16	0.185	0.006	0.086	34.660x
256	16	1	0.141	0.001	0.042	45.530x
256	16	2	0.168	0.000	0.068	38.302x
256	16	4	0.201	0.000	0.101	31.957x
256	16	8	0.207	0.000	0.108	30.937x
256	16	16	0.230	0.001	0.130	27.953x
256	32	1	0.124	0.000	0.025	51.739x
256	32	2	0.131	0.000	0.032	48.888x
256	32	4	0.158	0.001	0.058	40.734x
256	32	8	0.166	0.001	0.067	38.669x
256	32	16	0.186	0.002	0.087	34.454x
256	64	1	0.106	0.000	0.006	60.671x
256	64	2	0.119	0.000	0.020	53.881x
256	64	4	0.152	0.001	0.053	42.227x
256	64	8	0.156	0.002	0.056	41.179x
256	64	16	0.173	0.004	0.074	37.030x
256	128	1	0.101	0.000	0.001	63.758x
256	128	2	0.114	0.000	0.015	56.239x
256	128	4	0.153	0.001	0.054	41.858x
256	128	8	0.163	0.002	0.064	39.268x
256	128	16	0.190	0.006	0.091	33.773x
512	16	1	0.112	0.002	0.012	57.537x
512	16	2	0.132	0.000	0.032	48.788x
512	16	4	0.168	0.000	0.069	38.149x
512	16	8	0.165	0.001	0.065	39.004x
512	16	16	0.181	0.001	0.082	35.468x
512	32	1	0.106	0.000	0.006	60.745x
512	32	2	0.119	0.000	0.020	53.942x
512	32	4	0.151	0.001	0.052	42.506x
512	32	8	0.155	0.001	0.055	41.441x
512	32	16	0.171	0.002	0.071	37.641x
512	64	1	0.101	0.000	0.001	63.792x
512	64	2	0.114	0.000	0.014	56.482x
512	64	4	0.152	0.001	0.052	42.338x
512	64	8	0.156	0.002	0.057	41.161x

Continued on next page

Table A.24: Results of benchmarking mandelbrot using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	64	16	0.176	0.005	0.077	36.421x
1024	**16**	**1**	**0.100**	**0.001**	**0.000**	**64.326x**
1024	16	2	0.122	0.000	0.023	52.437x
1024	16	4	0.157	0.000	0.058	40.887x
1024	16	8	0.155	0.001	0.055	41.514x
1024	16	16	0.171	0.002	0.072	37.455x
1024	32	1	0.101	0.000	0.001	63.810x
1024	32	2	0.114	0.001	0.014	56.550x
1024	32	4	0.151	0.001	0.051	42.618x
1024	32	8	0.155	0.002	0.055	41.501x
1024	32	16	0.174	0.004	0.075	36.860x

A.4 Ridge3d

Table A.25: Results of benchmarking ridge3d using execution method "sequential"

mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
4.344	**0.021**	**4.327**	**1.000x**

Table A.26: Results of benchmarking ridge3d using execution method "cuda"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.851	0.001	0.833	5.105x
16	32	0.529	0.000	0.512	8.208x
16	64	0.262	0.000	0.244	16.606x
16	128	0.143	0.000	0.125	30.365x
16	256	0.096	0.000	0.078	45.287x
32	16	0.432	0.003	0.415	10.051x
32	32	0.248	0.000	0.230	17.528x
32	64	0.136	0.000	0.118	32.034x
32	128	0.084	0.000	0.067	51.455x
32	256	0.072	0.000	0.054	60.363x
64	16	0.203	0.002	0.186	21.369x
64	32	0.126	0.000	0.108	34.515x
64	64	0.074	0.000	0.056	58.709x
64	128	0.065	0.000	0.047	66.762x
64	256	0.058	0.000	0.040	74.999x
64	1024	0.042	0.000	0.025	102.805x
128	16	0.108	0.001	0.091	40.132x
128	32	0.072	0.000	0.054	60.478x
128	64	0.062	0.000	0.045	69.521x
128	128	0.055	0.000	0.038	78.477x
128	256	0.052	0.000	0.034	83.582x
128	512	0.041	0.001	0.024	104.824x
256	16	0.065	0.001	0.047	67.269x
256	32	0.064	0.000	0.047	67.616x
256	64	0.055	0.000	0.038	78.687x
256	128	0.067	0.000	0.049	65.287x
256	256	0.042	0.000	0.024	103.182x
512	16	0.060	0.001	0.042	72.762x
512	32	0.055	0.000	0.037	78.895x

Continued on next page

Table A.26: Results of benchmarking ridge3d using execution method "cuda" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	64	0.066	0.000	0.049	65.619x
512	**128**	**0.041**	**0.000**	**0.023**	**106.423x**
1024	16	0.060	0.001	0.043	71.807x
1024	32	0.066	0.000	0.048	66.134x
1024	64	0.042	0.001	0.024	104.477x

Table A.27: Results of benchmarking ridge3d using execution method "cuda-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.714	0.003	0.697	6.083x
16	32	0.427	0.001	0.409	10.182x
16	64	0.212	0.001	0.194	20.513x
16	128	0.111	0.000	0.093	39.267x
16	256	0.066	0.000	0.049	65.664x
32	16	0.359	0.003	0.342	12.096x
32	32	0.214	0.001	0.197	20.253x
32	64	0.107	0.001	0.089	40.618x
32	128	0.056	0.000	0.039	77.172x
32	256	0.039	0.001	0.022	110.248x
64	16	0.177	0.002	0.159	24.585x
64	32	0.105	0.001	0.087	41.548x
64	64	0.053	0.000	0.036	81.763x
64	128	0.034	0.001	0.016	129.229x
64	256	0.042	0.000	0.024	103.897x
128	16	0.088	0.001	0.071	49.262x
128	32	0.053	0.000	0.036	81.667x
128	64	0.034	0.001	0.016	129.298x
128	128	0.037	0.000	0.020	116.236x
128	256	0.042	0.000	0.024	104.228x
256	16	0.047	0.001	0.030	91.806x
256	32	0.033	0.001	0.015	131.920x
256	64	0.038	0.000	0.020	114.734x
256	128	0.041	0.000	0.023	106.340x
512	**16**	**0.032**	**0.000**	**0.014**	**135.260x**
512	32	0.037	0.000	0.019	117.357x
512	64	0.041	0.000	0.023	106.581x

Continued on next page

Table A.27: Results of benchmarking ridge3d using execution method "cuda-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
1024	16	0.034	0.000	0.017	127.194x
1024	32	0.041	0.000	0.023	106.672x

Table A.28: Results of benchmarking ridge3d using execution method "cuda-batch"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.706	0.003	0.688	6.158x
16	32	0.422	0.000	0.405	10.286x
16	64	0.210	0.000	0.192	20.707x
16	128	0.110	0.000	0.092	39.548x
16	256	0.067	0.000	0.049	65.100x
32	16	0.354	0.002	0.336	12.284x
32	32	0.209	0.000	0.191	20.817x
32	64	0.107	0.000	0.090	40.552x
32	128	0.057	0.000	0.039	76.433x
32	256	0.040	0.001	0.022	109.841x
64	16	0.175	0.003	0.157	24.878x
64	32	0.105	0.000	0.087	41.389x
64	64	0.054	0.000	0.036	80.739x
64	128	0.034	0.001	0.016	127.892x
64	256	0.032	0.000	0.014	135.370x
128	16	0.090	0.001	0.072	48.531x
128	32	0.054	0.000	0.036	80.640x
128	64	0.034	0.001	0.017	126.832x
128	128	0.031	0.000	0.013	142.249x
128	256	0.031	0.000	0.013	140.492x
256	16	0.050	0.001	0.032	87.519x
256	32	0.034	0.001	0.016	128.173x
256	**64**	**0.029**	**0.000**	**0.012**	**148.026x**
256	128	0.048	0.000	0.030	90.935x
512	16	0.033	0.000	0.015	131.597x
512	32	0.029	0.000	0.012	147.949x
512	64	0.048	0.001	0.030	90.535x
1024	16	0.039	0.000	0.022	110.313x
1024	32	0.047	0.000	0.030	91.597x

Table A.29: Results of benchmarking ridge3d using execution method "cuda-batch-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.714	0.004	0.696	6.084x
16	32	0.427	0.001	0.409	10.175x
16	64	0.212	0.001	0.195	20.473x
16	128	0.111	0.001	0.093	39.281x
16	256	0.066	0.000	0.049	65.659x
32	16	0.359	0.003	0.342	12.096x
32	32	0.214	0.001	0.197	20.276x
32	64	0.107	0.001	0.090	40.548x
32	128	0.056	0.000	0.039	77.126x
32	256	0.039	0.001	0.021	111.664x
64	16	0.177	0.003	0.159	24.578x
64	32	0.104	0.001	0.087	41.586x
64	64	0.053	0.000	0.035	81.900x
64	128	0.034	0.001	0.016	129.202x
64	256	0.042	0.000	0.024	103.898x
128	16	0.088	0.002	0.070	49.299x
128	32	0.053	0.000	0.036	81.744x
128	64	0.034	0.001	0.016	129.232x
128	128	0.037	0.000	0.020	116.188x
128	256	0.042	0.000	0.024	104.067x
256	16	0.048	0.002	0.031	90.020x
256	32	0.033	0.001	0.015	132.119x
256	64	0.038	0.000	0.020	114.633x
256	128	0.041	0.000	0.023	106.225x
512	**16**	**0.032**	**0.000**	**0.015**	**135.231x**
512	32	0.037	0.000	0.019	117.309x
512	64	0.041	0.000	0.023	106.605x
1024	16	0.034	0.000	0.016	127.328x
1024	32	0.041	0.000	0.023	106.610x

Table A.30: Results of benchmarking ridge3d using execution method "cuda-unified-memory"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	0.857	0.002	0.839	5.071x
16	32	0.535	0.001	0.517	8.123x
16	64	0.262	0.004	0.245	16.553x

Continued on next page

Table A.30: Results of benchmarking ridge3d using execution method "cuda-unified-memory" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	128	0.149	0.001	0.131	29.177x
16	256	0.102	0.000	0.084	42.601x
32	16	0.439	0.003	0.421	9.904x
32	32	0.254	0.000	0.237	17.081x
32	64	0.140	0.002	0.122	31.118x
32	128	0.091	0.001	0.073	47.996x
32	256	0.077	0.000	0.059	56.346x
64	16	0.210	0.001	0.192	20.686x
64	32	0.132	0.001	0.115	32.788x
64	64	0.086	0.001	0.069	50.359x
64	128	0.071	0.000	0.053	61.509x
64	256	0.063	0.001	0.046	68.741x
128	16	0.115	0.002	0.098	37.682x
128	32	0.079	0.000	0.061	55.178x
128	64	0.069	0.001	0.052	62.538x
128	128	0.060	0.001	0.043	72.143x
128	**256**	**0.057**	**0.000**	**0.040**	**75.955x**
256	16	0.071	0.001	0.053	61.224x
256	32	0.070	0.001	0.053	61.923x
256	64	0.061	0.001	0.043	71.444x
256	128	0.072	0.001	0.055	60.057x
512	16	0.065	0.001	0.047	66.797x
512	32	0.060	0.001	0.042	72.326x
512	64	0.068	0.001	0.051	63.459x
1024	16	0.067	0.001	0.049	65.137x
1024	32	0.072	0.000	0.055	60.219x

Table A.31: Results of benchmarking ridge3d using execution method "cuda-global-queue"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.617	0.003	0.600	7.037x
16	16	2	0.638	0.000	0.621	6.807x
16	16	4	0.648	0.000	0.630	6.706x
16	16	8	0.646	0.000	0.628	6.724x
16	16	16	0.648	0.000	0.631	6.702x

Continued on next page

Table A.31: Results of benchmarking ridge3d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	32	1	0.372	0.000	0.354	11.676x
16	32	2	0.383	0.000	0.366	11.328x
16	32	4	0.389	0.000	0.371	11.178x
16	32	8	0.384	0.000	0.366	11.325x
16	32	16	0.387	0.000	0.369	11.225x
16	64	1	0.184	0.000	0.167	23.577x
16	64	2	0.190	0.000	0.173	22.831x
16	64	4	0.193	0.000	0.175	22.538x
16	64	8	0.192	0.000	0.174	22.667x
16	64	16	0.195	0.000	0.177	22.276x
16	128	1	0.094	0.000	0.076	46.307x
16	128	2	0.097	0.000	0.080	44.574x
16	128	4	0.100	0.000	0.082	43.593x
16	128	8	0.100	0.000	0.083	43.384x
16	128	16	0.103	0.000	0.085	42.208x
16	256	1	0.055	0.000	0.037	79.114x
16	256	2	0.058	0.000	0.040	75.256x
16	256	4	0.060	0.000	0.042	72.939x
16	256	8	0.061	0.000	0.043	71.677x
16	256	16	0.063	0.000	0.045	69.026x
32	16	1	0.300	0.003	0.282	14.486x
32	16	2	0.310	0.000	0.292	14.018x
32	16	4	0.315	0.000	0.297	13.812x
32	16	8	0.314	0.000	0.296	13.842x
32	16	16	0.315	0.000	0.297	13.792x
32	32	1	0.181	0.000	0.164	23.957x
32	32	2	0.187	0.000	0.169	23.245x
32	32	4	0.190	0.000	0.172	22.920x
32	32	8	0.187	0.000	0.170	23.211x
32	32	16	0.189	0.000	0.171	22.972x
32	64	1	0.091	0.000	0.073	47.724x
32	64	2	0.094	0.000	0.076	46.230x
32	64	4	0.095	0.000	0.078	45.567x
32	64	8	0.095	0.000	0.077	45.803x
32	64	16	0.097	0.000	0.079	44.910x
32	128	1	0.048	0.000	0.030	91.248x

Continued on next page

Table A.31: Results of benchmarking ridge3d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	128	2	0.049	0.000	0.032	87.857x
32	128	4	0.051	0.000	0.033	85.788x
32	128	8	0.051	0.000	0.033	85.194x
32	128	16	0.053	0.000	0.035	82.488x
32	256	1	0.033	0.001	0.016	130.813x
32	256	2	0.035	0.000	0.018	123.345x
32	256	4	0.036	0.000	0.019	119.724x
32	256	8	0.037	0.000	0.019	117.426x
32	256	16	0.038	0.000	0.021	112.961x
64	16	1	0.149	0.001	0.131	29.198x
64	16	2	0.153	0.001	0.136	28.335x
64	16	4	0.156	0.000	0.138	27.934x
64	16	8	0.155	0.000	0.137	28.019x
64	16	16	0.156	0.000	0.138	27.880x
64	32	1	0.090	0.000	0.073	48.146x
64	32	2	0.093	0.000	0.075	46.725x
64	32	4	0.094	0.000	0.077	46.054x
64	32	8	0.093	0.000	0.076	46.593x
64	32	16	0.095	0.000	0.077	45.952x
64	64	1	0.047	0.000	0.029	93.268x
64	64	2	0.048	0.000	0.030	90.500x
64	64	4	0.049	0.000	0.031	88.951x
64	64	8	0.049	0.000	0.031	89.313x
64	64	16	0.050	0.000	0.032	87.246x
64	128	1	0.029	0.001	0.011	152.417x
64	128	2	0.030	0.000	0.012	144.294x
64	128	4	0.031	0.000	0.013	139.861x
64	128	8	0.031	0.000	0.014	138.227x
64	128	16	0.033	0.000	0.015	131.820x
64	256	1	0.021	0.000	0.003	209.797x
64	256	2	0.022	0.000	0.004	199.277x
64	256	4	0.023	0.000	0.005	191.249x
64	256	8	0.023	0.000	0.006	186.261x
64	256	16	0.025	0.000	0.007	174.339x
128	16	1	0.075	0.002	0.058	57.567x
128	16	2	0.077	0.000	0.060	56.219x

Continued on next page

Table A.31: Results of benchmarking ridge3d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	16	4	0.078	0.000	0.061	55.383x
128	16	8	0.078	0.000	0.061	55.561x
128	16	16	0.079	0.000	0.061	54.985x
128	32	1	0.046	0.000	0.029	93.737x
128	32	2	0.048	0.000	0.030	90.915x
128	32	4	0.049	0.000	0.031	89.416x
128	32	8	0.048	0.000	0.031	89.840x
128	32	16	0.049	0.000	0.032	87.918x
128	64	1	0.028	0.001	0.011	153.703x
128	64	2	0.030	0.000	0.012	145.862x
128	64	4	0.031	0.000	0.013	141.654x
128	64	8	0.031	0.000	0.013	140.374x
128	64	16	0.032	0.000	0.015	133.945x
128	128	1	0.021	0.000	0.003	210.059x
128	128	2	0.022	0.000	0.004	200.821x
128	128	4	0.022	0.000	0.005	193.105x
128	128	8	0.023	0.000	0.005	187.955x
128	128	16	0.024	0.000	0.007	178.022x
128	**256**	**1**	**0.018**	**0.000**	**0.000**	**246.511x**
128	256	2	0.019	0.001	0.001	231.245x
128	256	4	0.020	0.000	0.002	220.668x
128	256	8	0.021	0.000	0.003	211.607x
128	256	16	0.023	0.000	0.005	192.141x
256	16	1	0.044	0.001	0.026	99.339x
256	16	2	0.043	0.001	0.025	100.981x
256	16	4	0.042	0.000	0.025	102.504x
256	16	8	0.042	0.000	0.025	102.485x
256	16	16	0.043	0.001	0.026	100.384x
256	32	1	0.028	0.001	0.011	153.748x
256	32	2	0.030	0.000	0.012	146.577x
256	32	4	0.030	0.000	0.013	142.470x
256	32	8	0.031	0.000	0.013	141.444x
256	32	16	0.032	0.000	0.015	135.130x
256	64	1	0.021	0.000	0.003	211.233x
256	64	2	0.021	0.000	0.004	202.461x
256	64	4	0.022	0.000	0.005	194.195x

Continued on next page

Table A.31: Results of benchmarking ridge3d using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	64	8	0.023	0.000	0.005	189.951x
256	64	16	0.024	0.000	0.007	179.117x
256	128	1	0.018	0.000	0.000	241.092x
256	128	2	0.019	0.000	0.001	228.520x
256	128	4	0.020	0.000	0.002	218.169x
256	128	8	0.021	0.000	0.003	210.025x
256	128	16	0.023	0.000	0.005	192.589x
512	16	1	0.028	0.000	0.010	155.177x
512	16	2	0.029	0.000	0.011	151.413x
512	16	4	0.030	0.000	0.012	145.640x
512	16	8	0.031	0.000	0.013	142.427x
512	16	16	0.032	0.000	0.014	136.674x
512	32	1	0.020	0.000	0.003	212.184x
512	32	2	0.021	0.000	0.004	202.733x
512	32	4	0.022	0.000	0.005	195.283x
512	32	8	0.023	0.000	0.005	190.401x
512	32	16	0.024	0.000	0.007	178.493x
512	64	1	0.018	0.000	0.000	241.038x
512	64	2	0.019	0.000	0.001	229.067x
512	64	4	0.020	0.000	0.002	218.858x
512	64	8	0.021	0.000	0.003	209.950x
512	64	16	0.023	0.000	0.005	193.028x
1024	16	1	0.024	0.000	0.007	178.081x
1024	16	2	0.025	0.000	0.008	172.081x
1024	16	4	0.027	0.000	0.009	163.894x
1024	16	8	0.027	0.000	0.010	159.031x
1024	16	16	0.029	0.000	0.011	150.708x
1024	32	1	0.018	0.000	0.000	241.202x
1024	32	2	0.019	0.000	0.001	227.816x
1024	32	4	0.020	0.000	0.002	218.219x
1024	32	8	0.021	0.000	0.003	209.324x
1024	32	16	0.022	0.000	0.005	193.187x

Table A.32: Results of benchmarking ridge3d using execution method "cuda-global-queue-unified-memory"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.618	0.003	0.600	7.030x
16	16	2	0.639	0.000	0.621	6.800x
16	16	4	0.649	0.000	0.631	6.699x
16	16	8	0.647	0.000	0.629	6.716x
16	16	16	0.649	0.000	0.631	6.695x
16	32	1	0.373	0.000	0.355	11.659x
16	32	2	0.384	0.000	0.366	11.312x
16	32	4	0.389	0.000	0.372	11.163x
16	32	8	0.384	0.000	0.367	11.307x
16	32	16	0.387	0.000	0.370	11.213x
16	64	1	0.185	0.000	0.167	23.522x
16	64	2	0.191	0.000	0.173	22.782x
16	64	4	0.193	0.000	0.176	22.490x
16	64	8	0.192	0.000	0.174	22.626x
16	64	16	0.195	0.000	0.178	22.224x
16	128	1	0.094	0.000	0.077	46.123x
16	128	2	0.098	0.000	0.080	44.419x
16	128	4	0.100	0.000	0.082	43.459x
16	128	8	0.100	0.000	0.083	43.263x
16	128	16	0.103	0.000	0.086	42.094x
16	256	1	0.055	0.000	0.038	78.799x
16	256	2	0.058	0.000	0.040	75.048x
16	256	4	0.060	0.000	0.042	72.581x
16	256	8	0.061	0.000	0.043	71.478x
16	256	16	0.063	0.000	0.046	68.774x
32	16	1	0.300	0.001	0.282	14.482x
32	16	2	0.310	0.000	0.292	14.009x
32	16	4	0.315	0.000	0.297	13.799x
32	16	8	0.314	0.000	0.297	13.830x
32	16	16	0.315	0.000	0.298	13.780x
32	32	1	0.182	0.000	0.164	23.930x
32	32	2	0.187	0.000	0.170	23.200x
32	32	4	0.190	0.000	0.172	22.886x
32	32	8	0.187	0.000	0.170	23.179x
32	32	16	0.189	0.000	0.172	22.925x
32	64	1	0.091	0.000	0.074	47.597x

Continued on next page

Table A.32: Results of benchmarking ridge3d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	64	2	0.094	0.000	0.077	46.074x
32	64	4	0.096	0.000	0.078	45.433x
32	64	8	0.095	0.000	0.077	45.698x
32	64	16	0.097	0.000	0.079	44.781x
32	128	1	0.048	0.000	0.030	90.749x
32	128	2	0.050	0.000	0.032	87.417x
32	128	4	0.051	0.000	0.033	85.318x
32	128	8	0.051	0.000	0.034	84.868x
32	128	16	0.053	0.000	0.035	82.177x
32	256	1	0.031	0.000	0.013	141.038x
32	256	2	0.032	0.000	0.015	134.762x
32	256	4	0.033	0.000	0.016	130.999x
32	256	8	0.034	0.000	0.016	128.750x
32	256	16	0.035	0.000	0.018	123.350x
64	16	1	0.149	0.001	0.131	29.168x
64	16	2	0.154	0.000	0.136	28.265x
64	16	4	0.156	0.000	0.139	27.813x
64	16	8	0.156	0.000	0.138	27.912x
64	16	16	0.157	0.000	0.139	27.755x
64	32	1	0.091	0.000	0.073	47.917x
64	32	2	0.093	0.000	0.076	46.477x
64	32	4	0.095	0.000	0.077	45.831x
64	32	8	0.094	0.000	0.076	46.397x
64	32	16	0.095	0.000	0.077	45.735x
64	64	1	0.047	0.000	0.029	92.592x
64	64	2	0.048	0.000	0.031	89.800x
64	64	4	0.049	0.000	0.032	88.282x
64	64	8	0.049	0.000	0.031	88.518x
64	64	16	0.050	0.000	0.033	86.480x
64	128	1	0.028	0.001	0.011	154.472x
64	128	2	0.030	0.000	0.012	146.335x
64	128	4	0.030	0.001	0.012	144.925x
64	128	8	0.031	0.000	0.013	140.122x
64	128	16	0.031	0.001	0.013	139.635x
64	256	1	0.020	0.001	0.003	213.807x
64	256	2	0.022	0.000	0.004	197.048x

Continued on next page

Table A.32: Results of benchmarking ridge3d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	256	4	0.023	0.000	0.005	189.334x
64	256	8	0.024	0.000	0.006	184.371x
64	256	16	0.025	0.000	0.007	173.287x
128	16	1	0.076	0.001	0.058	57.499x
128	16	2	0.078	0.000	0.060	56.020x
128	16	4	0.079	0.000	0.061	55.217x
128	16	8	0.078	0.000	0.061	55.341x
128	16	16	0.079	0.000	0.062	54.839x
128	32	1	0.047	0.000	0.029	93.087x
128	32	2	0.048	0.000	0.030	90.466x
128	32	4	0.049	0.000	0.031	88.865x
128	32	8	0.049	0.000	0.031	89.294x
128	32	16	0.050	0.000	0.032	87.536x
128	64	1	0.028	0.001	0.011	153.702x
128	64	2	0.029	0.001	0.011	151.631x
128	64	4	0.029	0.000	0.012	148.561x
128	64	8	0.030	0.001	0.013	143.466x
128	64	16	0.031	0.000	0.013	140.796x
128	128	1	0.020	0.001	0.002	217.127x
128	128	2	0.022	0.000	0.004	198.609x
128	128	4	0.023	0.000	0.005	191.736x
128	128	8	0.023	0.000	0.006	187.513x
128	128	16	0.025	0.000	0.007	176.924x
128	**256**	**1**	**0.018**	**0.000**	**0.000**	**243.136x**
128	256	2	0.019	0.000	0.001	229.453x
128	256	4	0.020	0.000	0.002	218.926x
128	256	8	0.021	0.000	0.003	209.659x
128	256	16	0.023	0.000	0.005	189.915x
256	16	1	0.041	0.001	0.024	105.429x
256	16	2	0.042	0.000	0.024	103.800x
256	16	4	0.043	0.000	0.025	101.906x
256	16	8	0.043	0.000	0.025	101.766x
256	16	16	0.043	0.000	0.026	100.199x
256	32	1	0.026	0.000	0.009	164.943x
256	32	2	0.027	0.000	0.010	158.870x
256	32	4	0.029	0.001	0.011	149.279x

Continued on next page

Table A.32: Results of benchmarking ridge3d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	8	0.030	0.000	0.012	146.959x
256	32	16	0.030	0.000	0.013	144.095x
256	64	1	0.021	0.001	0.003	211.246x
256	64	2	0.022	0.000	0.004	199.830x
256	64	4	0.023	0.000	0.005	192.902x
256	64	8	0.023	0.000	0.005	188.414x
256	64	16	0.024	0.000	0.007	177.817x
256	128	1	0.018	0.000	0.001	238.377x
256	128	2	0.019	0.000	0.002	225.874x
256	128	4	0.020	0.000	0.002	216.639x
256	128	8	0.021	0.001	0.003	206.869x
256	128	16	0.023	0.000	0.005	190.648x
512	16	1	0.028	0.000	0.011	154.206x
512	16	2	0.029	0.000	0.011	149.472x
512	16	4	0.030	0.000	0.012	146.739x
512	16	8	0.029	0.001	0.012	147.473x
512	16	16	0.030	0.000	0.012	146.064x
512	32	1	0.020	0.001	0.002	222.122x
512	32	2	0.022	0.000	0.004	200.315x
512	32	4	0.023	0.000	0.005	192.748x
512	32	8	0.023	0.000	0.005	188.868x
512	32	16	0.025	0.000	0.007	176.895x
512	64	1	0.018	0.000	0.001	237.181x
512	64	2	0.019	0.000	0.002	225.514x
512	64	4	0.020	0.000	0.002	216.692x
512	64	8	0.021	0.000	0.003	207.882x
512	64	16	0.023	0.000	0.005	191.043x
1024	16	1	0.025	0.000	0.007	175.508x
1024	16	2	0.026	0.000	0.008	169.212x
1024	16	4	0.027	0.000	0.009	161.916x
1024	16	8	0.028	0.000	0.010	157.261x
1024	16	16	0.029	0.000	0.011	150.080x
1024	32	1	0.018	0.000	0.001	236.615x
1024	32	2	0.019	0.000	0.002	225.170x
1024	32	4	0.020	0.000	0.002	216.223x
1024	32	8	0.021	0.000	0.003	207.879x

Continued on next page

Table A.32: Results of benchmarking ridge3d using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
1024	32	16	0.023	0.000	0.005	191.749x

A.5 Vr-lite-cam

Table A.33: Results of benchmarking vr-lite-cam using execution method "sequential"

mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
6.415	**0.026**	**6.394**	**1.000x**

Table A.34: Results of benchmarking vr-lite-cam using execution method "cuda"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	2.237	0.002	2.217	2.867x
16	32	1.426	0.002	1.405	4.499x
16	64	0.817	0.001	0.797	7.849x
16	128	0.598	0.002	0.577	10.727x
16	256	0.485	0.000	0.464	13.236x
32	16	1.123	0.002	1.102	5.714x
32	32	0.705	0.001	0.685	9.094x
32	64	0.456	0.001	0.436	14.060x
32	128	0.329	0.001	0.308	19.498x
32	256	0.261	0.000	0.241	24.560x
64	16	0.596	0.002	0.575	10.767x
64	32	0.404	0.001	0.383	15.889x
64	64	0.264	0.001	0.244	24.285x
64	128	0.185	0.001	0.164	34.694x
64	256	0.156	0.000	0.135	41.220x
64	1024	0.077	0.002	0.057	83.070x
128	16	0.347	0.000	0.326	18.500x
128	32	0.249	0.000	0.229	25.744x
128	64	0.160	0.001	0.140	39.975x
128	128	0.126	0.000	0.105	51.080x
128	256	0.110	0.000	0.089	58.270x
128	512	0.077	0.001	0.056	83.300x
256	16	0.215	0.001	0.195	29.789x
256	32	0.151	0.000	0.131	42.436x
256	64	0.109	0.000	0.089	58.755x
256	128	0.096	0.000	0.075	66.770x
256	256	0.077	0.000	0.056	83.472x
512	16	0.138	0.000	0.117	46.465x

Continued on next page

Table A.34: Results of benchmarking vr-lite-cam using execution method "cuda" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	32	0.107	0.001	0.086	60.010x
512	64	0.092	0.001	0.071	69.936x
512	128	0.077	0.000	0.056	83.391x
1024	16	0.102	0.000	0.081	63.168x
1024	32	0.092	0.000	0.071	70.109x
1024	**64**	**0.076**	**0.000**	**0.056**	**83.858x**

Table A.35: Results of benchmarking vr-lite-cam using execution method "cuda-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1.319	0.006	1.298	4.864x
16	32	0.812	0.004	0.792	7.898x
16	64	0.455	0.002	0.434	14.103x
16	128	0.295	0.002	0.274	21.763x
16	256	0.226	0.001	0.205	28.429x
32	16	0.660	0.004	0.640	9.716x
32	32	0.405	0.002	0.384	15.850x
32	64	0.227	0.001	0.206	28.271x
32	128	0.150	0.001	0.129	42.904x
32	256	0.117	0.001	0.096	54.807x
64	16	0.335	0.004	0.314	19.161x
64	32	0.205	0.001	0.185	31.254x
64	64	0.117	0.001	0.096	54.994x
64	128	0.078	0.001	0.057	82.359x
64	256	0.064	0.000	0.043	100.670x
128	16	0.173	0.003	0.152	37.140x
128	32	0.114	0.001	0.093	56.237x
128	64	0.075	0.001	0.054	86.038x
128	128	0.060	0.000	0.039	107.590x
128	256	0.065	0.000	0.044	98.767x
256	16	0.096	0.002	0.075	66.881x
256	32	0.070	0.001	0.050	91.032x
256	64	0.052	0.000	0.031	123.787x
256	128	0.049	0.000	0.028	131.556x
512	16	0.062	0.001	0.042	102.716x
512	32	0.051	0.000	0.031	125.216x

Continued on next page

Table A.35: Results of benchmarking vr-lite-cam using execution method "cuda-permute" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
512	64	0.049	0.002	0.028	130.790x
1024	16	0.052	0.002	0.032	122.846x
1024	**32**	**0.048**	**0.001**	**0.028**	**133.165x**

Table A.36: Results of benchmarking vr-lite-cam using execution method "cuda-batch"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1.796	0.003	1.775	3.572x
16	32	1.150	0.001	1.130	5.577x
16	64	0.646	0.001	0.625	9.929x
16	128	0.537	0.002	0.517	11.937x
16	256	0.474	0.000	0.454	13.522x
32	16	0.916	0.003	0.896	7.000x
32	32	0.509	0.000	0.489	12.591x
32	64	0.342	0.001	0.321	18.773x
32	128	0.306	0.001	0.285	20.973x
32	256	0.239	0.000	0.218	26.848x
64	16	0.446	0.003	0.425	14.396x
64	32	0.272	0.000	0.251	23.612x
64	64	0.197	0.001	0.176	32.636x
64	128	0.151	0.001	0.130	42.564x
64	256	0.136	0.000	0.115	47.304x
128	16	0.237	0.002	0.217	27.050x
128	32	0.168	0.000	0.148	38.099x
128	64	0.108	0.000	0.087	59.517x
128	128	0.098	0.000	0.077	65.372x
128	256	0.073	0.000	0.052	87.991x
256	16	0.139	0.002	0.119	46.107x
256	32	0.097	0.000	0.076	66.469x
256	64	0.073	0.000	0.053	87.411x
256	128	0.065	0.000	0.045	98.373x
512	16	0.083	0.001	0.062	77.345x
512	32	0.069	0.000	0.048	93.389x
512	**64**	**0.056**	**0.000**	**0.035**	**114.747x**
1024	16	0.060	0.001	0.039	107.207x

Continued on next page

Table A.36: Results of benchmarking vr-lite-cam using execution method "cuda-batch" (Continued)

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
1024	32	0.056	0.000	0.035	114.357x

Table A.37: Results of benchmarking vr-lite-cam using execution method "cuda-batch-permute"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1.318	0.007	1.298	4.866x
16	32	0.813	0.003	0.793	7.888x
16	64	0.454	0.002	0.434	14.123x
16	128	0.294	0.001	0.274	21.784x
16	256	0.226	0.001	0.205	28.419x
32	16	0.660	0.004	0.639	9.725x
32	32	0.405	0.002	0.385	15.828x
32	64	0.228	0.001	0.207	28.186x
32	128	0.149	0.001	0.129	42.950x
32	256	0.117	0.001	0.096	54.880x
64	16	0.335	0.004	0.314	19.162x
64	32	0.205	0.001	0.184	31.348x
64	64	0.117	0.001	0.096	54.903x
64	128	0.078	0.001	0.057	82.382x
64	256	0.064	0.000	0.043	100.670x
128	16	0.173	0.002	0.152	37.136x
128	32	0.114	0.001	0.094	56.177x
128	64	0.074	0.000	0.054	86.160x
128	128	0.060	0.001	0.039	107.540x
128	256	0.065	0.000	0.044	98.580x
256	16	0.096	0.002	0.075	66.876x
256	32	0.070	0.001	0.050	91.292x
256	64	0.052	0.000	0.031	123.817x
256	**128**	**0.049**	**0.000**	**0.028**	**131.562x**
512	16	0.063	0.002	0.042	102.492x
512	32	0.051	0.000	0.031	125.135x
512	64	0.050	0.002	0.029	129.021x
1024	16	0.051	0.000	0.031	125.180x
1024	32	0.049	0.002	0.029	130.562x

Table A.38: Results of benchmarking vr-lite-cam using execution method "cuda-unified-memory"

n	b	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	2.386	0.005	2.365	2.689x
16	32	1.607	0.006	1.587	3.991x
16	64	1.125	0.008	1.104	5.703x
16	128	0.821	0.009	0.800	7.815x
16	256	0.701	0.008	0.680	9.153x
32	16	1.247	0.005	1.226	5.146x
32	32	0.850	0.005	0.829	7.548x
32	64	0.786	0.013	0.765	8.160x
32	128	0.627	0.012	0.606	10.235x
32	256	0.542	0.010	0.521	11.841x
64	16	0.756	0.006	0.736	8.483x
64	32	0.625	0.007	0.605	10.256x
64	64	0.682	0.011	0.662	9.401x
64	128	0.549	0.016	0.529	11.680x
64	256	0.519	0.011	0.498	12.363x
128	16	0.598	0.008	0.577	10.726x
128	32	0.726	0.021	0.705	8.836x
128	64	0.581	0.014	0.560	11.048x
128	128	0.560	0.015	0.540	11.449x
128	**256**	**0.478**	**0.008**	**0.457**	**13.426x**
256	16	0.618	0.020	0.598	10.375x
256	32	0.635	0.021	0.615	10.096x
256	64	0.626	0.014	0.606	10.244x
256	128	0.564	0.016	0.543	11.383x
512	16	0.565	0.014	0.545	11.344x
512	32	0.648	0.017	0.627	9.901x
512	64	0.585	0.018	0.564	10.969x
1024	16	0.582	0.016	0.561	11.030x
1024	32	0.507	0.012	0.486	12.661x

Table A.39: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.544	0.003	0.523	11.793x
16	16	2	0.575	0.000	0.554	11.154x
16	16	4	0.646	0.000	0.625	9.934x

Continued on next page

Table A.39: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	8	0.724	0.000	0.704	8.858x
16	16	16	0.843	0.001	0.822	7.610x
16	32	1	0.309	0.000	0.289	20.747x
16	32	2	0.346	0.000	0.325	18.563x
16	32	4	0.430	0.000	0.409	14.918x
16	32	8	0.517	0.001	0.496	12.407x
16	32	16	0.572	0.001	0.551	11.221x
16	64	1	0.165	0.000	0.144	38.858x
16	64	2	0.204	0.000	0.183	31.483x
16	64	4	0.267	0.000	0.246	24.026x
16	64	8	0.319	0.001	0.299	20.084x
16	64	16	0.350	0.002	0.329	18.330x
16	128	1	0.095	0.000	0.074	67.606x
16	128	2	0.149	0.000	0.129	43.013x
16	128	4	0.205	0.001	0.184	31.334x
16	128	8	0.238	0.001	0.217	26.953x
16	128	16	0.254	0.002	0.233	25.288x
16	256	1	0.077	0.000	0.056	83.188x
16	256	2	0.134	0.000	0.114	47.711x
16	256	4	0.186	0.000	0.165	34.524x
16	256	8	0.200	0.001	0.180	32.006x
16	256	16	0.209	0.001	0.188	30.731x
32	16	1	0.272	0.002	0.251	23.576x
32	16	2	0.288	0.000	0.267	22.294x
32	16	4	0.323	0.000	0.303	19.838x
32	16	8	0.363	0.001	0.343	17.658x
32	16	16	0.424	0.001	0.403	15.130x
32	32	1	0.155	0.000	0.135	41.300x
32	32	2	0.174	0.000	0.153	36.943x
32	32	4	0.216	0.000	0.196	29.662x
32	32	8	0.260	0.000	0.240	24.635x
32	32	16	0.292	0.000	0.272	21.958x
32	64	1	0.084	0.000	0.063	76.511x
32	64	2	0.104	0.000	0.083	61.970x
32	64	4	0.136	0.000	0.115	47.284x
32	64	8	0.162	0.001	0.142	39.489x

Continued on next page

Table A.39: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	64	16	0.178	0.001	0.158	35.997x
32	128	1	0.049	0.000	0.028	131.302x
32	128	2	0.076	0.000	0.056	83.962x
32	128	4	0.105	0.000	0.084	61.240x
32	128	8	0.123	0.001	0.102	52.228x
32	128	16	0.133	0.001	0.112	48.383x
32	256	1	0.044	0.001	0.023	147.285x
32	256	2	0.069	0.001	0.049	92.400x
32	256	4	0.096	0.001	0.075	66.869x
32	256	8	0.105	0.001	0.084	61.169x
32	256	16	0.189	0.001	0.168	33.995x
64	16	1	0.138	0.002	0.117	46.461x
64	16	2	0.146	0.000	0.125	44.004x
64	16	4	0.164	0.000	0.143	39.127x
64	16	8	0.185	0.000	0.164	34.713x
64	16	16	0.217	0.001	0.196	29.611x
64	32	1	0.080	0.000	0.059	80.575x
64	32	2	0.089	0.000	0.068	72.306x
64	32	4	0.111	0.000	0.090	58.009x
64	32	8	0.134	0.000	0.113	47.958x
64	32	16	0.153	0.001	0.132	42.030x
64	64	1	0.045	0.001	0.025	141.246x
64	64	2	0.054	0.001	0.034	118.100x
64	64	4	0.071	0.000	0.050	90.637x
64	64	8	0.086	0.001	0.065	74.706x
64	64	16	0.092	0.001	0.072	69.470x
64	128	1	0.029	0.001	0.008	223.876x
64	128	2	0.045	0.000	0.024	143.313x
64	128	4	0.056	0.001	0.035	114.623x
64	128	8	0.067	0.001	0.046	96.123x
64	128	16	0.118	0.002	0.098	54.181x
64	256	1	0.024	0.001	0.004	265.004x
64	256	2	0.041	0.000	0.021	154.768x
64	256	4	0.056	0.001	0.035	115.551x
64	256	8	0.103	0.001	0.083	62.151x
64	256	16	0.189	0.001	0.168	33.924x

Continued on next page

Table A.39: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
128	16	1	0.071	0.001	0.051	90.122x
128	16	2	0.077	0.000	0.057	82.879x
128	16	4	0.089	0.000	0.069	71.910x
128	16	8	0.102	0.000	0.081	63.092x
128	16	16	0.122	0.001	0.101	52.743x
128	32	1	0.045	0.001	0.025	142.113x
128	32	2	0.052	0.001	0.032	122.757x
128	32	4	0.067	0.000	0.046	95.711x
128	32	8	0.082	0.000	0.061	78.641x
128	32	16	0.092	0.000	0.071	69.994x
128	64	1	0.028	0.001	0.007	230.753x
128	64	2	0.041	0.000	0.020	157.406x
128	64	4	0.051	0.001	0.030	126.602x
128	64	8	0.061	0.001	0.040	105.381x
128	64	16	0.089	0.001	0.068	72.037x
128	128	1	0.021	0.001	0.001	299.800x
128	128	2	0.035	0.000	0.015	182.192x
128	128	4	0.049	0.002	0.028	130.645x
128	128	8	0.077	0.001	0.057	82.979x
128	128	16	0.133	0.002	0.113	48.084x
128	256	1	0.022	0.001	0.002	288.969x
128	256	2	0.038	0.000	0.017	170.495x
128	256	4	0.055	0.001	0.034	117.589x
128	256	8	0.103	0.001	0.082	62.483x
128	256	16	0.189	0.001	0.168	33.940x
256	16	1	0.042	0.000	0.021	152.498x
256	16	2	0.047	0.001	0.026	137.791x
256	16	4	0.054	0.000	0.033	119.158x
256	16	8	0.064	0.001	0.043	100.111x
256	16	16	0.077	0.001	0.056	83.691x
256	32	1	0.027	0.001	0.006	237.369x
256	32	2	0.039	0.000	0.018	163.957x
256	32	4	0.051	0.002	0.031	125.444x
256	32	8	0.058	0.001	0.038	109.931x
256	32	16	0.082	0.000	0.062	78.061x
256	64	1	0.021	0.001	0.000	309.035x

Continued on next page

Table A.39: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	64	2	0.034	0.000	0.013	188.270x
256	64	4	0.047	0.001	0.027	135.310x
256	64	8	0.065	0.001	0.044	98.946x
256	64	16	0.096	0.001	0.076	66.490x
256	128	1	0.022	0.001	0.001	293.318x
256	128	2	0.039	0.001	0.019	162.802x
256	128	4	0.051	0.002	0.030	125.588x
256	128	8	0.084	0.001	0.064	76.051x
256	128	16	0.134	0.002	0.113	47.869x
512	16	1	0.026	0.000	0.005	247.950x
512	16	2	0.035	0.000	0.014	183.721x
512	16	4	0.046	0.001	0.025	139.979x
512	16	8	0.050	0.001	0.030	127.342x
512	16	16	0.074	0.000	0.053	86.896x
512	**32**	**1**	**0.021**	**0.001**	**0.000**	**310.926x**
512	32	2	0.034	0.000	0.013	190.079x
512	32	4	0.047	0.001	0.026	136.687x
512	32	8	0.062	0.001	0.041	103.532x
512	32	16	0.085	0.000	0.064	75.475x
512	64	1	0.021	0.001	0.000	305.708x
512	64	2	0.038	0.001	0.018	167.868x
512	64	4	0.050	0.002	0.030	127.147x
512	64	8	0.069	0.000	0.048	93.586x
512	64	16	0.096	0.001	0.075	66.750x
1024	16	1	0.022	0.000	0.002	288.100x
1024	16	2	0.035	0.000	0.015	182.436x
1024	16	4	0.049	0.001	0.029	130.359x
1024	16	8	0.056	0.001	0.035	115.373x
1024	16	16	0.074	0.000	0.054	86.312x
1024	32	1	0.021	0.001	0.000	309.935x
1024	32	2	0.037	0.000	0.017	172.297x
1024	32	4	0.051	0.002	0.030	126.565x
1024	32	8	0.062	0.000	0.042	102.810x
1024	32	16	0.085	0.000	0.064	75.557x

Table A.40: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue-unified-memory"

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
16	16	1	0.577	0.003	0.556	11.122x
16	16	2	0.613	0.002	0.592	10.473x
16	16	4	0.685	0.002	0.665	9.360x
16	16	8	0.768	0.003	0.747	8.358x
16	16	16	0.894	0.003	0.873	7.178x
16	32	1	0.352	0.002	0.331	18.226x
16	32	2	0.388	0.002	0.368	16.515x
16	32	4	0.480	0.003	0.459	13.368x
16	32	8	0.593	0.007	0.573	10.812x
16	32	16	0.676	0.011	0.655	9.494x
16	64	1	0.257	0.003	0.237	24.924x
16	64	2	0.320	0.005	0.299	20.072x
16	64	4	0.418	0.008	0.397	15.362x
16	64	8	0.547	0.012	0.526	11.728x
16	64	16	0.613	0.023	0.592	10.466x
16	128	1	0.239	0.005	0.219	26.809x
16	128	2	0.319	0.009	0.298	20.130x
16	128	4	0.438	0.011	0.417	14.650x
16	128	8	0.582	0.025	0.562	11.015x
16	128	16	0.623	0.024	0.602	10.298x
16	256	1	0.254	0.006	0.233	25.297x
16	256	2	0.360	0.010	0.339	17.816x
16	256	4	0.502	0.015	0.482	12.770x
16	256	8	0.600	0.019	0.580	10.685x
16	256	16	0.594	0.020	0.574	10.791x
32	16	1	0.321	0.003	0.301	19.955x
32	16	2	0.345	0.004	0.324	18.620x
32	16	4	0.390	0.003	0.369	16.467x
32	16	8	0.433	0.006	0.412	14.812x
32	16	16	0.497	0.007	0.477	12.897x
32	32	1	0.242	0.004	0.221	26.506x
32	32	2	0.278	0.005	0.258	23.035x
32	32	4	0.323	0.006	0.302	19.873x
32	32	8	0.399	0.014	0.379	16.065x
32	32	16	0.469	0.023	0.449	13.674x
32	64	1	0.236	0.004	0.216	27.151x

Continued on next page

Table A.40: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
32	64	2	0.295	0.005	0.274	21.780x
32	64	4	0.381	0.012	0.361	16.825x
32	64	8	0.531	0.023	0.510	12.081x
32	64	16	0.580	0.029	0.560	11.057x
32	128	1	0.235	0.005	0.215	27.273x
32	128	2	0.305	0.007	0.284	21.028x
32	128	4	0.424	0.014	0.403	15.127x
32	128	8	0.578	0.024	0.558	11.090x
32	128	16	0.610	0.024	0.589	10.520x
32	256	1	0.251	0.006	0.230	25.577x
32	256	2	0.347	0.009	0.326	18.495x
32	256	4	0.480	0.012	0.459	13.376x
32	256	8	0.584	0.018	0.563	10.993x
32	256	16	0.629	0.016	0.608	10.203x
64	16	1	0.243	0.004	0.222	26.419x
64	16	2	0.279	0.006	0.258	22.989x
64	16	4	0.308	0.005	0.287	20.849x
64	16	8	0.337	0.009	0.316	19.051x
64	16	16	0.368	0.011	0.348	17.422x
64	**32**	**1**	**0.227**	**0.004**	**0.207**	**28.207x**
64	32	2	0.267	0.005	0.247	24.002x
64	32	4	0.304	0.007	0.283	21.119x
64	32	8	0.370	0.014	0.349	17.358x
64	32	16	0.434	0.026	0.414	14.766x
64	64	1	0.229	0.004	0.208	28.016x
64	64	2	0.279	0.006	0.259	22.958x
64	64	4	0.352	0.012	0.331	18.220x
64	64	8	0.498	0.017	0.477	12.884x
64	64	16	0.575	0.028	0.554	11.158x
64	128	1	0.234	0.005	0.214	27.380x
64	128	2	0.304	0.008	0.283	21.135x
64	128	4	0.424	0.013	0.404	15.117x
64	128	8	0.571	0.020	0.551	11.232x
64	128	16	0.610	0.017	0.589	10.521x
64	256	1	0.247	0.005	0.226	26.008x
64	256	2	0.349	0.010	0.328	18.379x

Continued on next page

Table A.40: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
64	256	4	0.479	0.011	0.459	13.379x
64	256	8	0.584	0.016	0.563	10.991x
64	256	16	0.626	0.015	0.605	10.246x
128	16	1	0.235	0.004	0.214	27.303x
128	16	2	0.274	0.005	0.253	23.452x
128	16	4	0.305	0.006	0.285	21.017x
128	16	8	0.358	0.013	0.338	17.898x
128	16	16	0.410	0.022	0.390	15.630x
128	32	1	0.229	0.005	0.209	27.985x
128	32	2	0.272	0.006	0.251	23.599x
128	32	4	0.325	0.010	0.304	19.746x
128	32	8	0.432	0.019	0.412	14.838x
128	32	16	0.510	0.028	0.489	12.581x
128	64	1	0.231	0.004	0.210	27.761x
128	64	2	0.285	0.006	0.265	22.485x
128	64	4	0.377	0.011	0.356	17.012x
128	64	8	0.528	0.025	0.508	12.138x
128	64	16	0.606	0.027	0.585	10.594x
128	128	1	0.239	0.005	0.219	26.808x
128	128	2	0.319	0.009	0.298	20.134x
128	128	4	0.450	0.015	0.429	14.255x
128	128	8	0.583	0.021	0.562	11.004x
128	128	16	0.620	0.022	0.600	10.342x
128	256	1	0.246	0.005	0.225	26.064x
128	256	2	0.349	0.009	0.328	18.390x
128	256	4	0.459	0.015	0.439	13.965x
128	256	8	0.571	0.013	0.550	11.233x
128	256	16	0.626	0.016	0.605	10.248x
256	16	1	0.235	0.005	0.215	27.276x
256	16	2	0.279	0.005	0.258	23.015x
256	16	4	0.333	0.009	0.312	19.283x
256	16	8	0.446	0.025	0.426	14.379x
256	16	16	0.526	0.032	0.506	12.185x
256	32	1	0.232	0.004	0.211	27.677x
256	32	2	0.292	0.006	0.271	21.980x
256	32	4	0.395	0.012	0.374	16.254x

Continued on next page

Table A.40: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
256	32	8	0.551	0.023	0.531	11.632x
256	32	16	0.619	0.030	0.599	10.356x
256	64	1	0.240	0.005	0.219	26.764x
256	64	2	0.322	0.011	0.301	19.932x
256	64	4	0.450	0.016	0.429	14.267x
256	64	8	0.603	0.025	0.583	10.633x
256	64	16	0.621	0.023	0.601	10.326x
256	128	1	0.252	0.006	0.231	25.474x
256	128	2	0.361	0.010	0.340	17.782x
256	128	4	0.515	0.020	0.494	12.455x
256	128	8	0.595	0.020	0.575	10.778x
256	128	16	0.611	0.018	0.590	10.498x
512	16	1	0.240	0.004	0.220	26.676x
512	16	2	0.293	0.006	0.272	21.919x
512	16	4	0.376	0.013	0.355	17.077x
512	16	8	0.518	0.025	0.497	12.396x
512	16	16	0.564	0.030	0.543	11.374x
512	32	1	0.239	0.004	0.218	26.866x
512	32	2	0.318	0.009	0.298	20.155x
512	32	4	0.449	0.016	0.428	14.284x
512	32	8	0.587	0.015	0.566	10.935x
512	32	16	0.617	0.025	0.597	10.391x
512	64	1	0.246	0.006	0.226	26.030x
512	64	2	0.344	0.010	0.323	18.667x
512	64	4	0.489	0.019	0.468	13.126x
512	64	8	0.611	0.021	0.590	10.501x
512	64	16	0.631	0.023	0.611	10.161x
1024	16	1	0.246	0.005	0.226	26.025x
1024	16	2	0.312	0.007	0.291	20.564x
1024	16	4	0.410	0.013	0.389	15.641x
1024	16	8	0.533	0.032	0.513	12.028x
1024	16	16	0.558	0.030	0.538	11.492x
1024	32	1	0.247	0.006	0.226	25.958x
1024	32	2	0.353	0.013	0.333	18.163x
1024	32	4	0.513	0.023	0.492	12.507x
1024	32	8	0.595	0.020	0.575	10.775x

Continued on next page

Table A.40: Results of benchmarking vr-lite-cam using execution method "cuda-global-queue-unified-memory" (Continued)

n	b	c	mean time (μ) in s	stddev time (σ) in s	$\Delta_{\text{global min}}$ in s	Speed up over seq.
1024	32	16	0.622	0.027	0.601	10.315x

Bibliography

[1] AMD. Hip programming guide v4.5¶.

[2] BELL, N., AND HOBEROCK, J. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[3] BERNSTEIN, G. L., SHAH, C., LEMIRE, C., DEVITO, Z., FISHER, M., LEVIS, P., AND HANRAHAN, P. Ebb: A dsl for physical simulation on cpus and gpus. *ACM Trans. Graph. 35*, 2 (may 2016).

[4] CABRAL, B., AND LEEDOM, L. C. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), SIGGRAPH '93, Association for Computing Machinery, p. 263–270.

[5] CHIW, C. Implementing mathematical expressiveness in diderot. 223.

[6] CHIW, C., KINDLMANN, G., REPPY, J., SAMUELS, L., AND SELTZER, N. Diderot: A parallel dsl for image analysis and visualization. *SIGPLAN Not. 47*, 6 (jun 2012), 111–120.

[7] CHIW, C., KINDLMANN, G., REPPY, J., SAMUELS, L., AND SELTZER, N. Diderot: A parallel dsl for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, Association for Computing Machinery, p. 111–120.

[8] CHOI, H., CHOI, W., QUAN, T. M., HILDEBRAND, D. G., PFISTER, H., AND JEONG, W.-K. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE transactions on visualization and computer graphics 20*, 12 (2014), 2407–2416.

[9] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (San Francisco, CA, 2004), pp. 137–150.

[10] EBERLY, D. *Ridges in image and data analysis*, vol. 7. Springer Science & Business Media, 2012.

[11] GUPTA, P. Cuda refresher: The cuda programming model, Aug 2022.

[12] HARRIS, M. Unified memory for cuda beginners, Aug 2022.

[13] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 556–571.

[14] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not. 52*, 6 (jun 2017), 556–571.

[15] JARZĄBEK, Ł., AND CZARNUL, P. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *The Journal of Supercomputing 73*, 12 (June 2017), 5378–5401.

[16] KINDLMANN, G., CHIW, C., SELTZER, N., SAMUELS, L., AND REPPY, J. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2016), 867–876.

[17] KINDLMANN, G., WHITAKER, R., TASDIZEN, T., AND MOLLER, T. Curvature-based transfer functions for direct volume rendering: methods and applications. In *IEEE Visualization, 2003. VIS 2003.* (2003), pp. 513–520.

[18] LTD, C. S. Computecpp™ community edition.

[19] MANDELBROT, B. B. *The fractal geometry of nature*, vol. 1.

[20] MCCORMICK, P., INMAN, J., AHRENS, J., MOHD-YUSOF, J., ROTH, G., AND CUMMINS, S. Scout: a data-parallel programming language for graphics processors. *Parallel Computing 33*, 10-11 (2007), 648–662.

[21] PHONG, B. T. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (jun 1975), 311–317.

[22] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not. 48*, 6 (jun 2013), 519–530.

[23] SCHOLZ, S.-B. Single-assignment c — functional programming using imperative style. In *6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK* (1994), J. Glauert, Ed., University of East Anglia, Norwich, England, UK, pp. 211–2113.

[24] VALIANT, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (aug 1990), 103–111.