Analysis of High Level implementations for Recursive Methods on GPUs

Cao, Cheng
University of California, Berkeley
Berkeley, CA
bobcao3@berkeley.edu

Kalloor, Justin
University of California, Berkeley
Berkeley, CA
jkalloor3@berkeley.edu

***Abstract*—Higher level DSLs have allowed for performant computation on GPUs while providing enough abstraction to the user to avoid significant deployment overhead. However, the SIMD/SIMT model of programming still can encounter unexpected performance drops when trying to translate naively from CPU code. One example of these performance drops is branch divergence, and this failure is especially exacerbated by recursive methods, as the depth of these recursions can vary greatly between threads.**

This paper investigates ways to enable recursion and task oriented programming using the Taichi DSL. We first present different methods of accomplishing this task, and benchmark each. Utilizing Taichi's multiple back-end code-gen targets, we investigate the performance of recursion tasks on different back-ends.

We compile these results into a final model that automatically chooses the best implementation for a given user program. In our benchmarks, we see massive improvement in throughput over the naive implementation, up to 500%.

***Index Terms*—GPU, Task-oriented Programming, DSL, CUDA**

I. Introduction

General Purpose GPU Programming has boomed in recent years, and enabled a new era of massively parallel computation. With hundreds of available cores, properly designed tasks will see a massive performance boost as opposed to traditional CPU programming.

The power of GPUs comes from the Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) model. By running many threads with the same program, the hardware can share the instruction / control side hardware while duplicating the data-path hardware. Under this model, a single program gets duplicated across tens of thousands of threads. This model allows for massive throughput on a variety for tasks [6] .

However, this type of programming also brings about the need for a Domain Specific Language (DSL). Traditional Programming Languages are not sufficient to describe these intricacies of the SIMT and SIMD model.

Currently there are a lot of different DSLs and APIs out there to program GPUs: some are proprietary and specific to hardware, some are open standards that can run on multiple hardwares. These languages include CUDA C++, OpenCL C, GLSL used in OpenGL and Vulkan, HLSL used in DirectX, and Metal shaders used in Apple hardwares.

All of these languages follow a very similar paradigm: serial functions and programs are compiled into GPU "kernels", and some extra bits of host side code commands the GPU to launch different kernels with different amounts of threads. Kernels are essentially sequential programs that work as a single thread, while the same program gets duplicated to massive amounts of input data (thus SPMD). Due to the quirks of the GPU, traditional recursive programs can cause huge performance issues due to cache divergence and control flow divergence. Currently, only CUDA allows true recursion through dynamic parallelism, and programmers using other languages or non-Nvidia GPUs will need to manually write a stack themselves, or convert their algorithms to an iterative one.

The process of converting a recursive algorithm to a producer-consumer queue paradigm is nothing new, but doing so takes a lot of programmer time and allows more bugs and errors than necessary. There are ways to sort and reorder queues to achieve higher performance on GPUs, but there are few previous works that attempted to create a generalized optimization system. Additionally, this process requires the programmer to create multiple different kernels serving different functions, and they might need to split up their original recursive program into multiple kernels. This process greatly complicates the programming experience.

II. Our Contributions

In this paper we:

- Discuss how to utilize a global queue in order to enable task-oriented recursion using a high level DSL called Taichi.
- Present different methods of enabling task-oriented recursion.
- Compare these different implementations, analyzing their performance as well as their GPU resource utilization on different backend APIs
- Use our findings to create an optimized model that queries different parameters of the recursive method.

For this paper, our metrics of success are defined as:

- Enable a subset of recursive methods that is optimized for the GPU through a high level DSL.
- Outperform naive implementations of these same recursive methods in performance by eliminating divergence.

- Describe performance patterns we see on the GPU and add additional optimizations to further boost performance.

To this end, we were able to mostly succeed in our project. We develop a global queue implementation that allows for performant ray tracing and graph search methods. For Vulkan and Metal backends, we see performance gains across different ray tracing benchmarks. Additionally, we utilize these findings to create an optimized model for different ray tracing benchmarks, and discuss further optimizations in the final section.

III. Background

A. SIMT/SIMD Programming model

GPUs exploit SIMD pipelines in order to run highly parallel programs with massive throughput. The power comes form running a single instruction across a vector of data elements, where each element is processed with the same instruction. Because a single control path can drive a vector of data path, these types of programming models can run very efficiently [6]. Usually, not all of the threads are running in lock-step. A limited sized group called a warp will share a control path. The thread group size might differ based on the hardware, but for all GPUs, all threads on each group can only be running the same instruction, or they will need to idle to wait for their chance to use the control path.

NVIDIA, through the development of CUDA, decided to unify all forms of parallelism available on their GPUs with a base unit of work, a thread. They classified their programming model as SIMT, as they expose an fully sequential and scalar programming model instead of directly exposing the underlying vector hardware [1].

An important feature of SIMT programming is that the GPU hardware handles the parallel execution and thread management. This typically limits the user to create isolated threads, since the scheduling of threads are not user controllable and global atomic operations are expensive. [6].

With the SIMT architecture the programmer must be very careful. Introducing branches (if statements, while loops) can cause GPU branch divergence. Since the GPU schedules only one set of instructions for a block, branches can lead to issues, as certain threads will require one set of instructions, while others will require another. In order to compensate, the GPU will run all sets of instructions together, and then throw away any sets that are not chosen [2]. In addition, this may cause forward progression issues and therefore dead-locks. Some threads in the group are spin-waiting on the other group of threads, while those threads never get scheduled as they are on a different path of execution.

B. Recursive Programming

Recursive Programming is a method of find a solution using solutions to smaller instances of the same problem. It is largely defined by functions calling themselves, which then utilizes the function stack data structure to make progress.

Many recursive problems are seemingly parallel. For example, consider a large graph search. The recursive function calls itself on all of its neighbors. So distinct neighbors can run this recursive call in parallel, and leverage hardware such as the GPU to run this efficiently.

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

Fig. 1: CUDA C++ Programming Language (Embedded DSL)

Recursion is an example of data irregular workload for a GPU [3]. One issue with using GPUs to parallelize recursion is divergence. Different threads will recurse to different depths or call different functions. As certain nodes in the graph run out of neighbors to call, or if some nodes are calling different functions to compute the results, some of the threads will idle while the other threads in the same thread group finish. This causes a massive drop in throughput and diminishes the speedup gained from parallelization. Because recursion, is task parallel rather than the classic SIMD data parallel paradigm, some transformations must be applied in order to run [9].

C. Ray Tracing

Ray tracing is an algorithm that generates an image by simulating light by using the concept of rays and ray-scene intersection. To generate a photo-realistic image, people usually resort to path tracing which is a form of ray tracing that uses recursive Monte-Carlo integration to produce a final image. Because how the rendering equation is formulated, this problem is inherently recursive. This problem is not trivial either, usually requiring massive amount of samples to compute an single image.

D. Domain Specific Languages

Due to the various intricacies and quirks of the GPU hardware, there is a need for a simplifying compiler. These Domain Specific Languages (DSLs) allow users to write fairly simple code that can be executed efficiently by the underlying hardware. There are several examples of DSLs used for GPU programming including CUDA, OpenCL, OpenGL, Vulkan, just to name a few. These different DSLs vary in overhead and user complexity, but the main goal for all of them is to enable GPU programming. An small example of CUDA can be found in Figure 1.

E. Taichi

Taichi is a high-level parallel programming DSL that uses a auto-parallelization scheme to convert for loops into GPU kernels. Unlike the other DSLs, a single Taichi kernel can be a combination of multiple backend kernels as a single

program might change its degree of parallelization mid-way. For example, in the world of simulation, an material point method based simulation might take many steps while each step iterates on different grids of work. In a traditional graphics API, the programmer will need to manage multiple kernels and figure out how to pass values around these kernels. In addition, Taichi also natively supports sparse data structures and the ability to separately declare an algorithm and the data structure it is running on, allowing great flexibility. For our project, Taichi's ability to generate code on many backends, proprietary or vendor neutral, benefits us by allowing the same program and data structure to be tested on different backends and hardwares easily. We ran our program with OpenGL, CUDA, and Vulkan on Windows and Linux computers, and we also ran experiments on Macbooks on Apple's GPU. This gives us the ability to use multiple backends to test our theory, and we won't be limited by a specific backend's implementation.

IV. Related Work

A. Persistent Threads

The idea of overriding the native hardware scheduler is an idea that has shown up in several papers [4], [5], [12], most prominently with persistent threads. While traditional GPU programming requires the user to spawn many more threads than there are physical executors, persistent thread programming creates a small set of active threads. These threads are enough to fill the GPU hardware, while generally not enough for the GPU scheduler to interfere. Furthermore, and more importantly, there are work queues created for a block of threads to poll. When a block finishes, it checks the queue for more and more work. By designing the queue fetch and push policy, the user is able to control the order and location of execution of tasks [5]. These queues are very important to enable communication, so various techniques such as creating queues with unfailing atomic operations and large access techniques have been explored [11].

Due to this ability to override the hardware scheduler to some extent, the persistent thread model of programming can be used to implement recursion. Threads are able to dynamically add a variable amount of work to the queue and then relaunch. This queuing system avoids going back to the CPU for synchronization and to launch a new kernel (which is impossible in traditional GPU APIs, where only the host can launch more threads). Due to these advantages, it is seen that persistent threads outperform naive SIMD implementation for small workloads and normal deeply-recursive algorithms [5]. However, implementing these work queues in an efficient manner is still yet to be available at a high level DSL due to the complexity and control of the design.

B. GPU Scheduling enhancements

There is some work looking into optimizing divergent workloads, such as recursion, by optimizing the way SIMD GPUs schedules its instructions. Since all instructions must execute in serial within a work group, it becomes a sorting problem where the compiler is trying to find an serial order of all instructions on different threads within the same group. This method focuses on better scheduling and re-convergence schemes within a single work group [7]. In practice, there is a lot of optimization potential left in trying to do scheduling in a global level. We will detail one example of this in the next section.

C. Ray Tracing Performance

In addition to recursion, specific techniques for improving ray tracing on GPUs is a massively explored field. Simply taking the original CPU code and passing it to the GPU runs into many standard GPU complications: divergence, memory limits, scheduling, etc. In a paper by Laine, Karras, and Aila it is shown that splitting the wavefront kernel into several smaller kernels with local queues can help improve performance, especially if there are many active processes. [8].

Workloads like path tracing can have divergence issues due to different recursion depths and different material functions to evaluate during each task. Due to the nature of Monte Carlo simulation, a large amount of random tasks are generated. While they might be random, it is conceivable that there are still large amount of tasks running the same code. For example, once a ray intersects with an object, random values are used to generate out-going rays, and these rays may hit different objects causing a branch divergence. On the other hand, other samples from other locations may hit the same object, even if they are not originally from the same work group. In this case global scheduling can help significantly by grouping together similar work globally and thus mostly eliminating divergence.

V. Methods

In this paper, we focused on bench marking with ray tracing. We took the relatively well known program called ray-tracing-in-one-weekend by Peter Shirley [10]. It is not a full fledged path-tracer, but it is representative of a real program. Because it is relatively simple, we can modify it relatively easily, while still getting representative benchmark results.

A. Naive and Iterative Ray Tracing Program

The naive / base version of this program is listed here:

1: **for** pixel i, j on the image **do** ▷ Parallel for
2: **for** $sample$ in $0 \dots N$ **do**
3: $x \leftarrow Camera(i, j)$
4: $omega \leftarrow CameraRay(i, j)$
5: $color_{i,j} \leftarrow color_{i,j} + L_o(x, \omega)$
6: **end for**
7: **end for**

Where $Camera$ is the function that transforms the pixel i, j coordinate to a world space 3d coordinate, and $CameraRay$ is the function that generates a ray direction based on the sensor plane coordinate. The L_o function is the rendering equation, and it is evaluated with Monte-Carlo integration that can be described as following:

1: $\omega_o = RandomSample(x, \omega_i)$
2: $x_{next} = IntersectScene(x, \omega_o)$

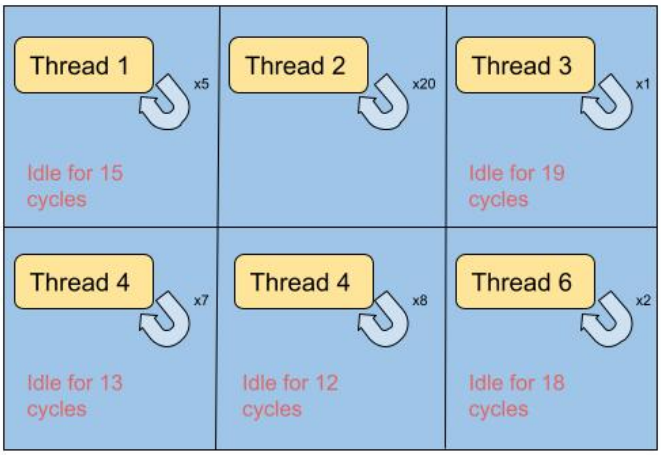

Fig. 2: Thread to task mapping of the Naive solution

if Has intersection **then**
    $L_o = EvaluateBrdf(x, \omega_i, \omega_o) * L_o(x_{next}, \omega_o)$
else
    $L_o = EvaluateSky(\omega_o)$
end if

Recursion is not traditionally supported on GPUs, however, naive path tracing is a tail recursive algorithm. So, this can be rewritten iteratively:

for pixel i, j on the image **do** ▷ Parallel for
    for $sample$ in $0 \dots N$ **do**
        $x \leftarrow Camera(i, j)$
        $omega \leftarrow CameraRay(i, j)$
        for $depth$ in $0 \dots maxrecursiondepth$ **do**
            ...
        end for
    end for
end for

In this paper we will call this version the "naive" solution. When this algorithm runs on the GPU, each pixel (i.e. each iteration of the outer most loop) gets allocated a thread, this can be illustrated by Figure 2.

This also means that we can just fold all the inner loops into one single loop. The inner most loop in the naive implementation can have different length as it is recursion based on random samples. This means when a group of pixel is launched, some of it may run long while the others are just idling. We theorize that on GPU, folding all inner loops can bring a decent performance uplift. We call this version the mega-loop, and it is structured like this:

for pixel i, j on the image **do** ▷ Parallel for
    $sample \leftarrow 0$
    $depth \leftarrow 0$
    $x \leftarrow Camera(i, j)$
    $omega \leftarrow CameraRay(i, j)$
    $throughput \leftarrow 1$
    while $sample < N$ **do**
        ...
        $throughput \leftarrow throughput \times EvaluateBrdf()$
        $depth \leftarrow depth + 1$
        if $depth \geq depth_{max}$ or no intersection **then**
            $color_{i,j} \leftarrow EvaluateSky() \times throughput$
            $sample \leftarrow sample + 1$
            $depth \leftarrow 0$
        end if
    end while
end for

B. Wave-front based Ray Tracing

Wave-front based ray tracing splits up the entire ray tracing workload into batches of tasks, where each task is called a wave-front. Usually each bounce is counted as the same wave-front, so that after each bounce, the wave front shrinks in size because some rays might not hit an object and gets terminated on the last wave-front. This approach theorizes that by doing things in wave-fronts, there are chances to sort the wave front and compact them to remove work items that we know will be just idling in a work group. Thus this approach is very similar to the mega-loop approach, but instead of having the GPU threads potentially waiting when we have threads that have exhausted their work, a new wave front is launched that compacts all thread groups. In essence, this approach pulled the inner while loop to the outer most layer.

for pixel i, j on the image **do** ▷ Parallel for
    $sample_{i,j} \leftarrow 0$
    $depth_{i,j} \leftarrow 0$
    $throughput_{i,j} \leftarrow 1$
    $x_{i,j} \leftarrow Camera(i, j)$
    $omega_{i,j} \leftarrow CameraRay(i, j)$
end for
while $sample_{i,j} < N$ **do** ▷ Serial while
    $pixels_{active} \leftarrow \forall sample_{i,j} < N$
    for i, j in $pixels_{active}$ **do** ▷ Parallel for
        ...
        $throughput_{i,j} \leftarrow throughput_{i,j} \times Brdf()$
        $depth_{i,j} \leftarrow depth + 1$
        if $depth_{i,j} \geq depth_{max}$ or no intersection **then**
            $color_{i,j} \leftarrow EvaluateSky() \times throughput_{i,j}$
            $sample_{i,j} \leftarrow sample_{i,j} + 1$
            $depth_{i,j} \leftarrow 0$
        end if
    end for
end while

Wave-front style ray-tracing is quite a bit more complex than the single kernel iteration based approach, as the programmer now needs to write multiple kernels, sequence them correctly, and also manage the interfaces where previously local variable now needs to live in global memory. In traditional DSLs and graphics APIs, this is quite painful, as each shader program only maps to a single parallel for, and a lot of host side code is needed to manage multiple programs like this. Fortunately in Taichi this is much simpler, as you can have many different parallel or serial kernels that forms a larger kernel.

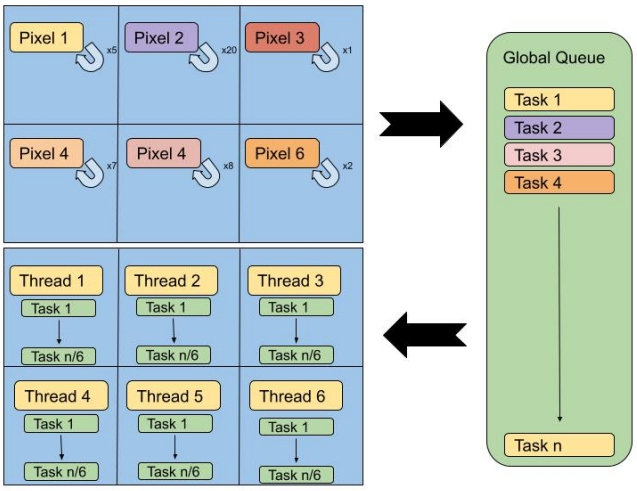

Fig. 3: Thread to task mapping of the Global Queue solution

C. Global Queue

This approach draws the same idea as the previous ones, but now it breaks the fundamental mapping between threads and pixels, where each thread may process different pixels. This is essentially the persistent thread model we have discussed earlier. A minimal group of threads is launched where it is enough to saturate the GPU pipeline while avoiding over subscribing the GPU's resources. Under this model, the problem truly transforms into a task oriented problem, where each bounce, each sample, and each pixel are all transformed into atomic (can not be further split) tasks. One task might spawn more task, such as the situation where a ray bounce hits an object. The benefit of this approach is that every thread is going to be fed with tasks, thus no idling. With sufficiently small kernels, each task will be mostly similar, thus reducing control flow divergence. Multiple queues can also be used to handle different situations, such as the if and else case of a branch. In this case, there will be no control flow divergence. This is achieved by breaking the sequential model of program, where each time a branch of function call happens, this call is buffered into a queue, where a bunch of coherent worker threads will process this task without divergence. We theorize that with enough memory bandwidth to back the queues, we should observe maximum performance with this model, while having almost full GPU utilization without wasting resources.

This can be illustrated by Figure 3.

Ray tracing workload using global queues can be described as:

for pixel i, j on the image **do** ▷ Parallel for
  $sample_{i,j} \leftarrow 0$
  $depth \leftarrow 0$
  $throughput \leftarrow 1$
  $x \leftarrow Camera(i, j)$
  $omega \leftarrow CameraRay(i, j)$
  Enqueue($\{depth, throughput, x, omega, i, j\}$)
end for
while Work available in queue **do** ▷ Parallel while
  $\{depth, throughput, x, omega, i, j\} \leftarrow$ Dequeue()
  ...
  $throughput \leftarrow throughput \times Brdf()$
  $depth \leftarrow depth + 1$
  if has intersection **then**
    Enqueue($\{depth, throughput, x, omega, i, j\}$)
  end if
  if $depth_{i,j} \geq depth_{max}$ or no intersection **then**
    $color_{i,j} \leftarrow EvaluateSky() \times throughput$
    $sample_{i,j} \leftarrow sample_{i,j} + 1$
    if $sample_{i,j} < N$ **then**
      $depth \leftarrow 0$
      $throughput \leftarrow 1$
      $x \leftarrow Camera(i, j)$
      $omega \leftarrow CameraRay(i, j)$
      Enqueue($\{depth, throughput, x, omega, i, j\}$)
    end if
  end if
end while

However, in practice this approach is quite tricky due to a few reasons. First, unless the parameters are fine tuned for a specific workload on a specific hardware, it is very hard to find an optimal amount of threads to fully utilize the GPU. We hope to come up with performance models that can help provide more insight into how to tune this parameter automatically. The second issue is the implementation difficulty, as the memory ordering are not well defined on many grpahics APIs and different backends may have very different behaviour. Locking is expensive on GPU, and without some care, it is very easy to fall into a deadlock due to the fact that GPU threads are executed in lock step within a work group. We will discuss these difficulties later.

D. Benchmarking Method

We converted an existing naive Taichi implementation of ray-tracing in one weekend to all of the methods we laid out above. Then we utilize Taichi's multi-backend capability to run experiments on multiple backends and multiple hardware devices. Mean while, we control the parameters of ray tracing, and we also introduced complexity parameterization in the scene intersection function, where we can make the task more or less complex. Using these controls, we measured the time spent to generate an image using various programs and parameters.

We hope for a performance scaling to verify our theories about the performance of various methods on GPU. We will capture traces where the hardware allows (e.g. using NSight to capture non-intrusive traces on Nvidia GPUs), and we will try to use the traces to either verify our theory or provide insights in case our theory does not match the actual behaviour.

VI. Findings

A. Benchmark Results

We set our benchmark parameters to an 2048×1024 output image, with 1, 8, and 32 samples per pixel, and with a maximum recursion depth of 32. Then, we ran the different methods (Naive, Mega-loop, Wave-front, Lock-less Global Queue) on all the different supported back-end on our RTX3080 GPU, we got some very intriguing results.

In the previous sections, we have mentioned that we hypothesize that the Naive approach will run the slowest, while Mega loop and wave front will be faster than Naive by about the same speedup. Additionally, we think the lock-less global queue will run fastest, if implemented correctly. We present our benchmark results in Figure 4, 5, and 6. In these results, the vertical axis is time spent on the workload (smaller is better), and we have different approaches paired with different back-ends listed on the horizontal axis. We don't have data running on OpenGL for both Wave-front and Lock-less Queue as the Taichi OpenGL backend lacks the required capabilities to run these two tests.

We can see that in all of these results, if we compare them horizontally across backends, OpenGL runs slower than both CUDA and Vulkan, especially when the total amount of tasks and complexity of the workload increase. (More samples means more possible divergence and re-convergence). CUDA in general out performs all the other back-ends consistently except in lock-less global queue, where it consistently falls behind Vulkan. To verify whether we can actually get better performance out of global queue, we ran these benchmarks on an Apple GPU (M1) so that we hope to isolate the problem. The performance on Apple's Metal API follows the same trend as Vulkan on the Nvidia RTX3080, thus we can confirm the strange behaviour here is limited to CUDA, and we will further analyze why this might be the case later.

If we compare the benchmark between different methods, a few odd things come into our view. We can see that the wave-front approach can be significantly slower than other approaches or even be slower than the naive approach, which is very different from our prediction. We collected traces of this program and we will present the reason why the performance of wave-front ray-tracing looks like what it is later on.

Another discovery while we are making the benchmarks, is that the parameters used in the lock-less global queue, especially the amount of concurrent threads to spawn, is very unpredictable. The optimal number changes from backend APIs to another, and it also depends on the complexity of the program in multiple ways (register pressure, memory access, total amount of compute in the tasks, etc.) The numbers here presented in the benchmarks are derived from the best parameters which are tested and hand-tuned, instead of basing on a heuristic. We find that this parameter influences the performance a lot, and it is especially hard to tune: spawning too little threads causes under utilization of GPU resources, spawning too many threads seem to run into contention and

Fig. 4: Performance measurements of $N = 1$

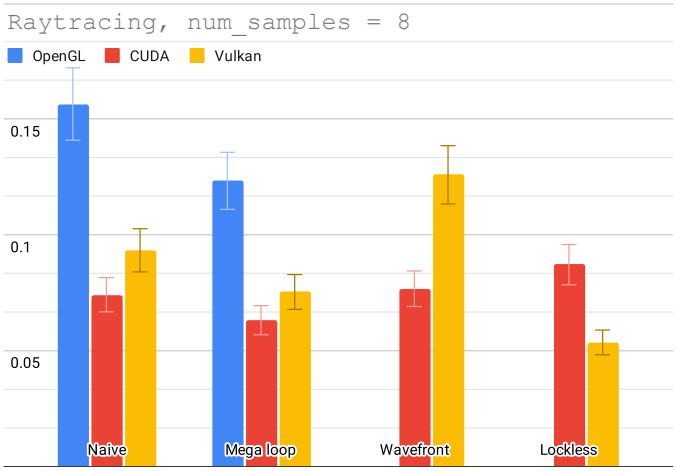

Fig. 5: Performance measurements of $N = 8$

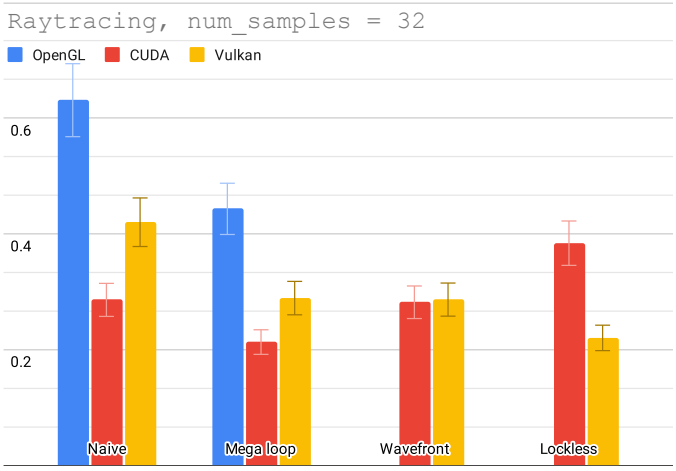

Fig. 6: Performance measurements of $N = 32$

forward progression issue, where it can slow down a lot or even entirely lock up.

In order to further investigate the possible explanations and solutions to the performance we observed, we ran GPU traces with the Vulkan back-end, with $N = 32$. The traces can be found in Figure 7, 8, 9, 10, 11, and 12.

B. Analysis of the Wave-front Approach

Previously we mentioned that the wave-front approach performed much worse than we expected. Therefore we run traces on both the mega-loop approach (Figure 7 and the wave-front approach 8, and we compared the traces. We made several discoveries:

1) The average GPU utilization when the kernel is active is higher in the wave-front approach. This is as expected as we stated that the compaction of wave-fronts before each dispatch makes the thread group unlikely to have idle threads. However it is still only maxing out at around 80%.
2) The wave-front approach have an extremely long tail where both the GPU utilization (gray) and active throughput (yellow) is low. While the mega-loop approach does have a tail, it is a relatively short tail.
3) If we zoom in to the individual wave-fronts (Figure 8), we can see it is like a much shorter version of the mega-loop trace, with some gaps in between. The gaps are caused by host communication where the host waits until a wave-front to finish, and then check whether it needs to launch the next wave-front. As for why there is still a tail in each wave-front, we suspect that is because the memory barrier causing a GPU pipeline flush, where the GPU waits for all pipelines to drain. Later we will propose a solution of the host side communication delay, but we are not able to come up with a solution for the pipeline draining behaviour.

In order to fix the host communication issue, we need a way for the GPU to dispatch more threads dynamically based on the results of previous dispatches, without the intervention of the host. This is called device-side enqueue, but it is currently only available in CUDA and OpenCL 2.1, which means it is not an commonly available API. The other option is to exhaustively and conservatively enqueue additional kernels, where you launch the maximum possible amount of wave fronts from the host and in the later wavefronts when there is no work, the kernel will immediately return. This is obviously still inefficient as you are allocating computing resources for kernels that would just immediately return, but at least it eliminates the need or greatly reduce the need of host-device communication. We benchmarked the conservative dispatch approach against the host side enqueue (Figure 10 and 11), and we found a performance improvement from around $500ms$ to the number we are showing in our previous benchmark results. (We are already showing the conservative dispatch version in the benchmark results section)

As for the long tail for the entire workload and pipeline draining behaviour between wave-fronts, we did not manage to find out a way to solve it. However the analysis on the global lock-less queue may give more insights onto this problem.

C. Analysis of the Global Lock-less Queue Approach

We ran trace for the global lock-less queue approach on Vulkan with $N = 32$. As mentioned before, in our benchmark data, Vulkan and Apple Metal shows a positive performance improvement of various scale. However, this is not the case for CUDA where we see performance degradation or even inconsistent performance.

We will first focus on the results from the Vulkan backend. If we look at Figure 12, we observe that:

1) The GPU resource allocation and active throughput tracks closely, which means we are achieving near theoretical level of throughput. This is in huge contrast to other versions where we always see a bit of under-utilization of allocated resources.
2) It has an extremely short tail, suggesting that all threads are still active till the last point. However this might be deceiving, as in the persistent thread model, until the queue is explicitly closed, all threads will spin-wait, and this will be interpreted as useful work by the profiler. This suggests an lack of appropriate tooling for persistent-thread style programming on the GPU.

In addition, when we are debugging the lock-less queue implementation, we encountered a lot of trouble. While the thread count is a fairly easy to sort out problem by running tests and finding the optimum, it is very hard to debug problems caused by GPU's execution model and memory consistency model. For example, some version of the code may lock-up when the thread count exceed a fairly arbitrary number. Sometimes we also observed that the program may work on one back-end but not on others albeit when they all run on the same GPU. We suspect the driver uses different memory model for different APIs and thus causing this inconsistency.

Troubles aside, we can still learn from the global lock-less queue as it does provide a decent performance benefit on APIs like Vulkan or Metal. The idea of decoupling the threads, delaying and regrouping tasks are key to achieve consistently high GPU usage.

VII. Model

With all of this data, it became clear that different the back-end global queue must change based on the given program. In order to fully realize this, we constructed a variable parameter model to select the best backend queue for performance. Using a basic ray tracing benchmark, we were able to vary the benchmark across different axes. We chose to vary the max depth of the pixel, the complexity of the hit() function, and the number of pixels. The max depth of the pixel is directly correlated to the depth of the recursion. The complexity of the hit() function increases the branch divergence of each ray tracing pixel. The number of pixels is linked to the number of parallel tasks able to be launched on the GPU. The results of these tests are displayed in Figure 13.

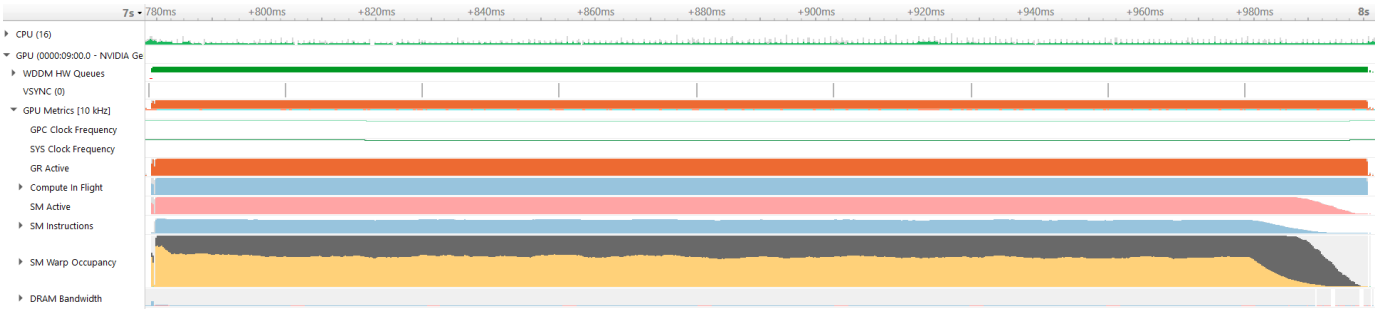

Fig. 7: GPU Utilization Trace of the Mega-loop approach

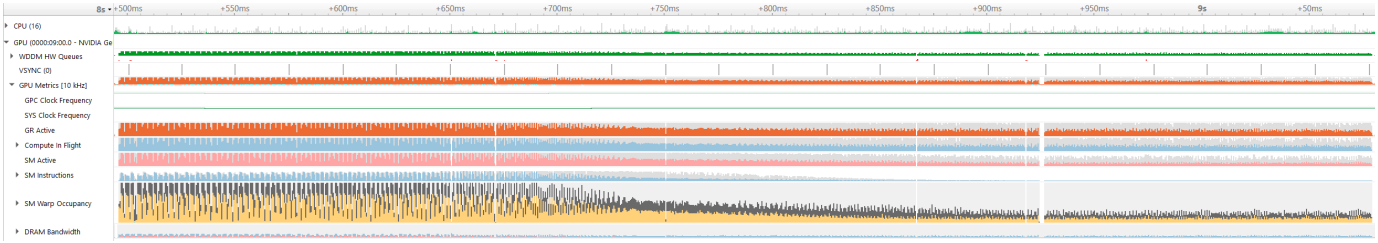

Fig. 8: GPU Utilization Trace of the Wave-front Approach

Fig. 9: GPU Utilization Trace of the Wave-front Approach, zoomed in on individual wave-fronts

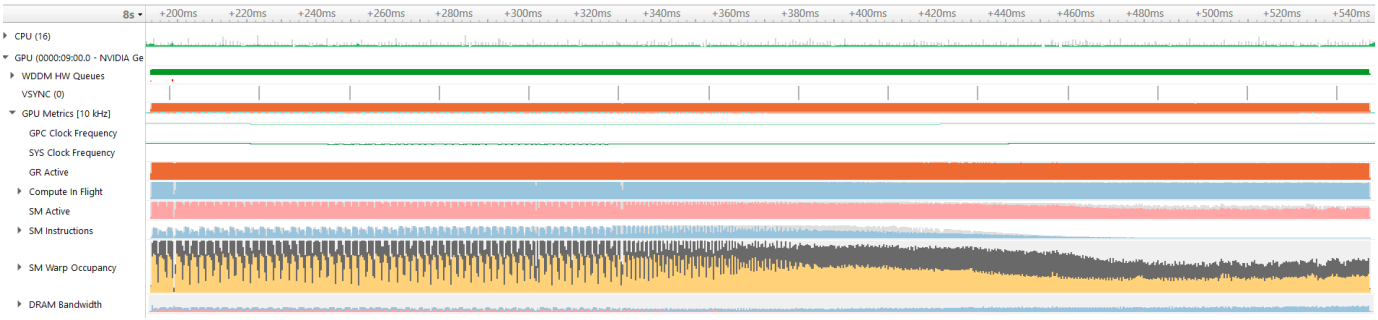

Fig. 10: GPU Utilization Trace of the Wave-front Approach with conservative dispatch

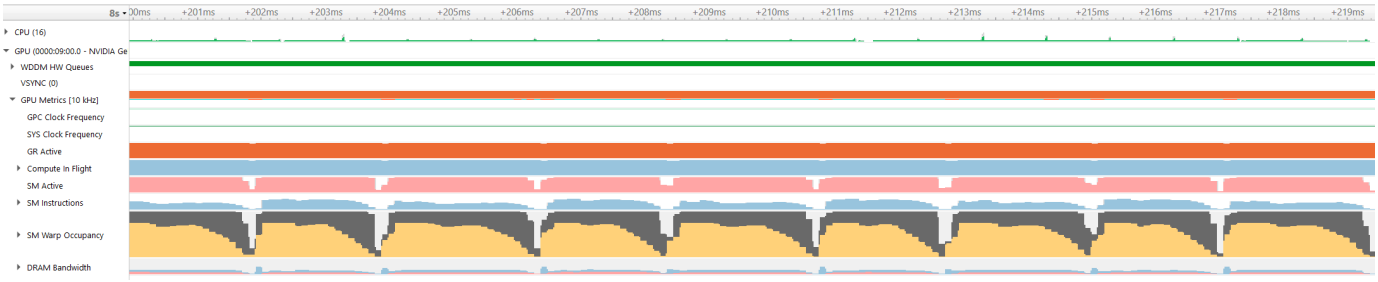

Fig. 11: GPU Utilization Trace of the Wave-front Approach with conservative dispatch, zoomed in on individual wave-fronts

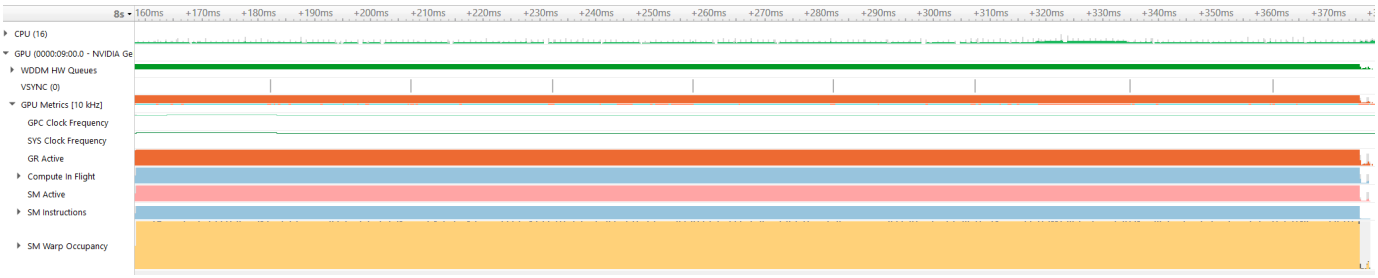

Fig. 12: GPU Utilization Trace of the Global Lock-less Queue approach

Based on these results, we constructed a K nearest neighbors model to predict the best queue for any particular task and launch that as a backend end. The performance is shown in figure 15. Note in this model, we were not able to run the lockless queue for larger sizes due to backend issues with Taichi, so that data is omitted.

VIII. Next Steps

Our solution scaled as expected for some backends (Vulkan, Metal) but clearly did not produce the expected results for CUDA. Our first next step in this research project is to fully understand the drop in performance for CUDA by tracing through the entire Taichi compilation.

Furthermore, the model and benchmarks were specifically run for ray tracing benchmarks. We want to extend this tool for more generic task-basked recursive tasks. We developed a basic tree search benchmark, but have not extended the tool more generally than that. Initial results from the tree search benchmark are shown in Table I.

Nodes	Taichi w/ locked queue	CPU
512	0.0281	0.0018
1024	0.0230	0.0019
2048	0.0221	0.0039
4096	0.0243	0.0089
8192	0.0250	0.0145
16384	0.0219	0.0281
32768	0.0227	0.0419

TABLE I: Graph search results outlining comparison between CPU recursive tree search vs a basic Taichi dynamic global queue. As seen, the GPU processing has additional overhead, but scales better for larger graphs.

For all the recursive benchmarks, there is an additional set of enhancements that can be implemented throughout the compilation process. These include ordering the global queue by some priority (while keeping it lockless) as well as implementing microkernels. Our model is developed such that choices for these lowering decisions can decided by the program parameters, which can offer even greater performance boosts.

IX. Conclusion

General Purpose GPU Programming is still a growing field, but the performance boosts offered by the GPU is unfortunately limited in its scope and requires low level understanding to manage seemingly simple tasks. Higher level tools help to remove this massive programmer overhead.

Our paper offers a solution to bridge a portion of this gap, by enabling recursion in a high level DSL called Taichi. We utilize principles of dynamic task queues to avoid GPU branch divergence. In Section V, we present different methods of enabling a simple ray tracing algorithm. These methods vary in complexity and difficulty in implementation, and also perform best in different with different parameters. Because of this variance, we develop a variational model to best enable performant ray tracing.

In the end, we were able to achieve our goals for this paper. We performed a detailed analysis on the various forms of ray tracing recursion, which all outperform the naive implementation. Furthermore, by creating a model, the user can simply specify a ray tracing function, and our tool can lower down to the best implementation for their particular use case.

Our ultimate goal is to enable all recursion on GPUs at a high level. This paper focused mainly on ray tracing, which showcases a useful example of a subset of this problem space. In order to fully support the full range of parallelizable

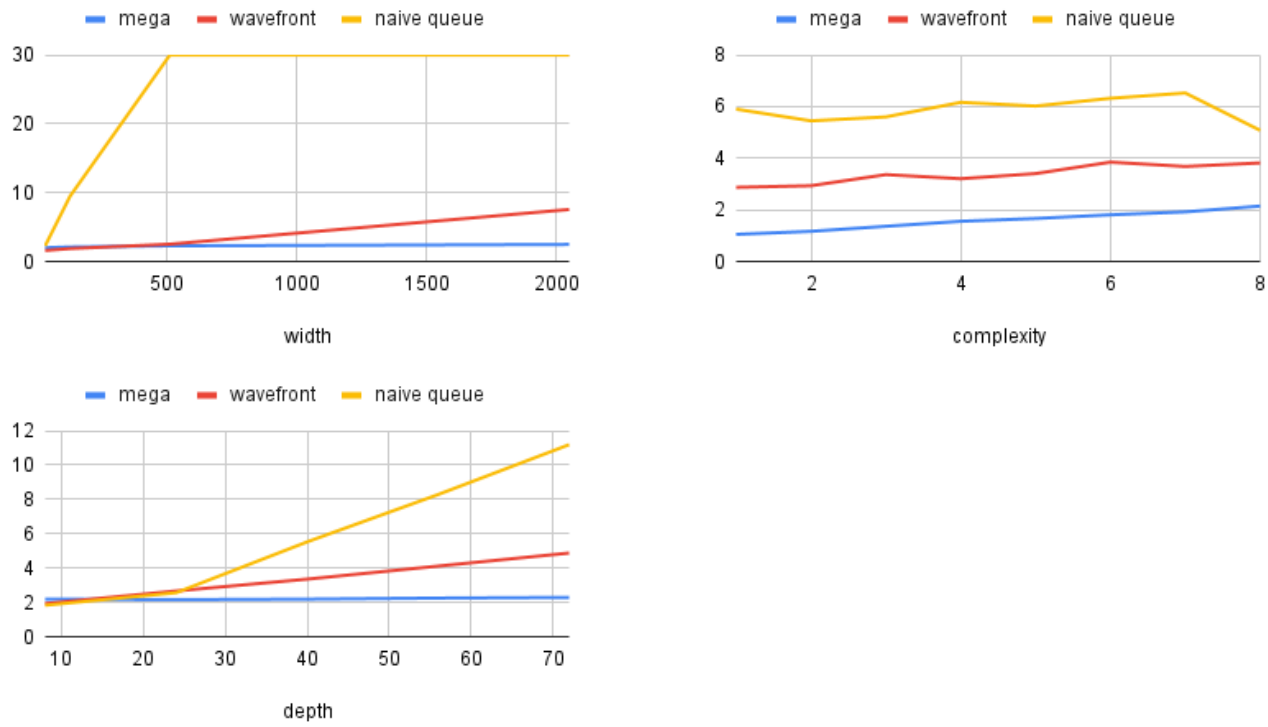

Fig. 13: Results of ray trace implementations using a parametrized ray trace benchmark and varying image width, max depth, and hit complexity.

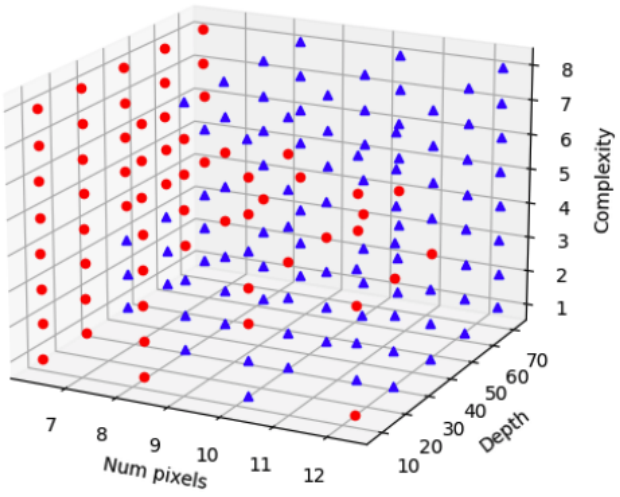

Fig. 14: KNN sample points while varying the number of pixels, complexity of the hit function and the maximum depth of the ray trace. Red points describe configurations where wavefront performed best, while the blue points represent the mega-loop implementation.

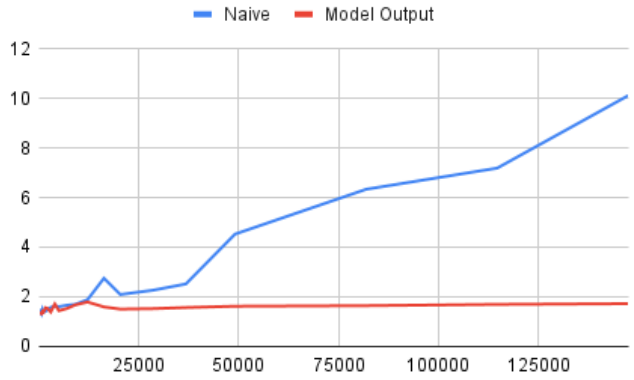

Fig. 15: Latency comparison of model vs naive approach as function of image width and depth. We see as much as 5x improvement as the image size scales up.

recursive problems, we expect additional structures to be necessary, such as a local function metadata stack. We hope that our findings here highlight the potential advantage of higher level GPU support and outline one method of realizing this advantage.

References

[1] "CUDA C++ Programming Guide," p. 441.

[2] J. Anantpur and G. R., "Taming Control Divergence in GPUs through Control Flow Linearization," in *Compiler Construction*, A. Cohen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 133–153.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.

[4] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Austin Texas USA: ACM, Jan. 2017, pp. 3–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3018743.3018748

[5] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads," in *2012 Innovative Parallel Computing (InPar)*. San Jose, CA, USA: IEEE, May 2012, pp. 1–14. [Online]. Available: http://ieeexplore.ieee.org/document/6339596/

[6] J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer architecture: a quantitative approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012, oCLC: ocn755102367.

[7] X. Huo, S. Krishnamoorthy, and G. Agrawal, "Efficient scheduling of recursive control flow on GPUs," in *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13*. Eugene, Oregon, USA: ACM Press, 2013, p. 409. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2464996.2479870

[8] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: wavefront path tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13*. Anaheim, California: ACM Press, 2013, p. 137. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2492045.2492060

[9] B. Ren, S. Balakrishna, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Extracting SIMD Parallelism from Recursive Task-Parallel Programs," *ACM Transactions on Parallel Computing*, vol. 6, no. 4, pp. 1–37, Dec. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3365663

[10] P. Shirley, "Ray tracing in one weekend," December 2020, `https://raytracing.github.io/books/RayTracingInOneWeekend.html`. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[11] D. Troendle, T. Ta, and B. Jang, "A Specialized Concurrent Queue for Scheduling Irregular Workloads on GPUs," in *Proceedings of the 48th International Conference on Parallel Processing*. Kyoto Japan: ACM, Aug. 2019, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3337821.3337837

[12] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach California USA: ACM, Jun. 2015, pp. 119–130. [Online]. Available: https://dl.acm.org/doi/10.1145/2751205.2751213