Formal Methods for GPGPU Programming: Is the Demand Met?

Lars B. van den Haak[1](✉), Anton Wijs[1], Mark van den Brand[1], and Marieke Huisman[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
{l.b.v.d.haak,a.j.wijs,m.g.j.v.d.brand}@tue.nl
[2] University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl

Abstract. Over the years, researchers have developed many formal method tools to support software development. However, hardly any studies are conducted to determine whether the actual problems developers encounter are sufficiently addressed. For the relatively young field of GPU programming, we would like to know whether the tools developed so far are sufficient, or whether some problems still need attention. To this end, we first look at what kind of problems programmers encounter in OpenCL and CUDA. We gather problems from Stack Overflow and categorise them with card sorting. We find that problems related to memory, synchronisation of threads, threads in general and performance are essential topics. Next, we look at (verification) tools in industry and research, to see how these tools addressed the problems we discovered. We think many problems are already properly addressed, but there is still a need for easy to use sound tools. Alternatively, languages or programming styles can be created, that allows for easier checking for soundness.

Keywords: GPU · GPGPU · Formal methods · Verification · Bugs · CUDA · OpenCL

1 Introduction

General-purpose GPU (GPGPU) programming has been around for over 10 years now, but is notoriously hard to do. In this work, we want to explore what kind of problems people experience during GPGPU programming and understand what the difficulties are in overcoming these problems. We accomplish this in two steps. First we find the problems and next we analyse current solutions in the domain of formal methods. We view this work as a way of identifying further research challenges and directions in this domain, with the aim to ease the difficulty of programming for a GPU.

To find the problems programmers encounter, we looked at Stack Overflow, which is a widely known website where programmers can ask questions related to programming. We took a sample of questions that are related to OpenCL and CUDA, the two dominant GPGPU programming languages, and categorise them using card sorting. These categories give us an up-to-date overview of (most) problems people encounter.

The next step is finding verification tools. Many tools have been developed that help people in their GPU programming work, like GPUVerify [11], Oclgrind [31], GKLEE [24], VerCors [12] and CUDA-MEMCHECK [1]. Although, only some of these have been picked up by developers of GPGPU programs. We look at scientific conferences and industry companies for tools. We narrow the scope to correctness issues and link the tools that solve these issues and indicate what improvements research can make.

In conclusion, in this work, we aim to help other researchers to focus their research on GPGPU programming problems that are not or incompletely addressed with and tools.

We make the following contributions.

1. An overview of common problems people struggle with whilst programming a GPGPU (Sect. 3).
2. Addressing problems of Sect. 3 where we think formal methods can make a direct contribution. We discuss solutions of existing tools and new research opportunities (Sect. 4).

2 Background

We base this section mostly on the CUDA Programming Guide [2]. GPUs are massive parallel compute devices, that work via the Single Instruction Multiple Threads (SIMT) execution model, which means that multiple threads are executing the same instruction in parallel, but with other data. In this paper, we consider mainly the CUDA and OpenCL programming languages. We work with the CUDA terms, but give the corresponding OpenCL terms in parentheses in this section. CUDA compiles to PTX [3], a pseudo-assembly language, which we call the *instruction level*, similarly OpenCL compiles to SPIR [4].

Functions that are executed on the GPU are called *kernels*. One can start kernels from the CPU, which we call the *host*. The GPU itself is called the *device*. Data stored on the RAM is not automatically accessible on the GPU and must be sent from the host to the device before invoking the kernel that uses the data. The programmer can schedule memory transfers and kernel executions in a queue.

Threads (Work-Items). When scheduling a kernel, you specify how many *threads* (*work-items*) are going to be executing this kernel. Threads are grouped together in *thread blocks* (*workgroups*) and all the thread blocks together form the *grid* (*NDRange*). From the hardware perspective, thread blocks are subdivided into *warps* (*sub-groups* or AMD calls them *wavefronts*), that typically have a size of 32 (64 on AMD devices) threads. Threads of a warp are executed in *lockstep*, meaning that they execute all the instruction at the same time.[1] If threads of a warp take different execution paths, e.g. due to if statements, the warp executes each path, but disables threads that are not on that path. This is called *thread divergence*, which can lead to performance loss.

A GPU consists of multiple *streaming multiprocessors*, which execute the warps in lockstep. Each thread block is assigned to one streaming multiprocessors.

Memory Model. A programmer has to manage the memory of a GPU manually. It has *global memory*, where transferred data from the host is stored, and any thread can access it. *Shared memory* (*local memory*) is shared in a thread block, which is faster than global memory. One can use it to share results within a thread block or to have faster access when data is reused. Per thread data is automatically stored in fast-access *registers*, or slow *local* memory in case not enough registers are available. For optimal global memory accesses, the accesses should be fully *coalesced*: this happens if threads of a warp call consecutive memory addresses and the first address is a multiple of the warp size.

Synchronization. When two threads do a read and write, or two writes to the same memory address, and this could happen simultaneously, this is called a *data race*. Data races lead to non-determinism and are considered, in most cases, a bug. A GPU can synchronize with a *barrier* on the thread block level, which ensures that all threads wait for each other before continuing execution. It also makes sure that after the synchronization, all writes to global and shared memory are performed, or depending on the barrier, only to shared memory. Thus, barriers can prevent *intra-block* data races in a thread block. All threads in a thread block must reach the same barrier, otherwise it results in undefined behaviour and is called *barrier divergence*.

In between threads of different thread blocks, synchronization is not possible with a (standard) global barrier, although Sorensen et al. [37] show how this can be constructed. Data races in between thread blocks are called *inter-block* data races. When lockstep execution of warps is not ensured also *intra-warp* data races can occur.

Synchronization can also be achieved via *fine-grained synchronization* using locks or atomics. Locks can make sure that only one thread has access to a specific memory address. Atomics allow for communication via memory, without risks of data races and GPUs typically implement them more efficiently than locks. A GPU has a weak memory model [6], which means that memory actions within a thread can be reordered by the hardware if there exist no dependencies within the thread. Therefore, when using fine-grained synchronization, specific memory actions may not yet be visible to other threads. Memory *fences* can be inserted to enforce a memory order, which might be needed to make sure that no *weak-memory* data races occur.

[1] Although this is not exactly true any more for Nvidia's Volta architecture and onward. See https://developer.nvidia.com/blog/inside-volta/.

Other Features. Some other features are less used, although we do want to mention them since they come up in this work. *Dynamic parallelism* allows parent kernels, to launch child kernels. A parent and child kernel have a consistent view of global memory at the start of the launch, but this is not guaranteed while executing. The parent kernel can synchronize with the child kernels it launched. A child kernel can recursively call a new child kernel. *Warp-level primitives* (*subgroup primitives*) are primitives that allow communication between threads in a warp, via the faster registers. For instance, one can use them to make a faster scan and reduction operation.

3 GPGPU Programming Problems

To know how formal methods can help solve GPGPU problems, we first need to know with what actual developers are struggling with. Therefore, we look at Stack Overflow, which is the go-to place for programming-related questions and is used by many programmers as a reference. Of the languages programmers use for GPGPU programming, CUDA (729 questions), OpenMP (471) and OpenCL (311) are the most popular, based on the number of question asked on Stack Overflow in 2019.[2] We focus on CUDA and OpenCL since OpenMP does not solely focusses on the GPU.

We first explain our approach for gathering and categorizing the results (Sect. 3.1). Next, we present the categories of programming problems we found, which we again ordered into themes and sub-themes for a clear overview (Sect. 3.2).

3.1 Approach

Gathering Problems. As argued above, we look at OpenCL and CUDA on Stack Overflow. Looking at the general tag `gpgpu`, `cuda` and `opencl`, we found that the 7 most related tags are `gpu`, `c++`, `nvidia`, `c`, `parallel-processing`, `thrust` and `nvcc`. The first five tags we consider too general, which would pollute our results. The tags `thrust` and `nvcc` are a specific CUDA library and compiler, which we do not want to focus on. Therefore, we stick with the tags `gpgpu`, `cuda` and `opencl`. On March 2, 2020 there are 17,539 questions on stack overflow that have the tag `cuda`, `opencl` or `gpgpu`.[3] We look at 376 Stack Overflow questions, which is a representative sample with a confidence level of 95% and a confidence interval of 5%s. Thus, with a 95% chance, we identify the problems which are present in at least 5% of the questions in the tags mentioned above.

Categorizing Problems. On the gathered questions, we performed open card sorting [27, Card-sorting: From Text To Themes], which creates categories in an unknown data set. We decided to look at the title, body and answers of the

[2] https://data.stackexchange.com/stackoverflow/query/1258739/gpgpu-tags.
[3] https://data.stackexchange.com/stackoverflow/query/1258838/gpgpu.

questions, to determine the categories. The first author, together with another PhD student, sorted the first 84 questions, where they achieved a mutual understanding of categories and held discussions for any corner cases. The next 43 cards were sorted separately, but in the same room, which allowed discussion on difficult cards. Eventually, this led to 26 different categories. The last 260 cards were sorted alone by the first author, and we ended up with 34 categories. For cards we could sort in multiple categories, we made new overlapping category or sorted them to the most appropriate category. After the sorting, we went over the relevant questions once more, to see if a newly made category would be more suitable.

Relevant Problems for Formal Methods. In the 34 categories, we make two distinctions. First, we mark problems that are interesting for GPGPU programming; these are 28 of the 34 categories. The non-relevant categories are related to (GPU) hardware, errors in the host code (unrelated to CUDA or OpenCL API calls), installing the correct CUDA and OpenCL drivers or libraries, setting up a development environment, linking libraries and questions related to OpenGL. In total, we found that 220 of the 376 questions were relevant to GPGPU programming.

We present the 28 GPGPU categories in the remainder of this section. We mark the ones (10) where we think formal methods are directly applicable to solve correctness problems underlying these questions.

3.2 Results

The results of the card sort are visible in Fig. 1. To organize the results, we have put some structure into them. We identified two themes: *memory* and *threads and synchronization*. We place the remaining categories in the *general* theme. Within each theme, we distinguish between bugs and performance-related questions as sub-themes. The results of this can be viewed in Fig. 2. We will explain each theme with its associated categories in the following subsections.

Memory. We first consider the *bugs* sub-theme categories: ‘memory transfer bug’, ‘out of bounds’ and ‘memory bug’. An *out of bounds* error occurs when an array is indexed outside its bounds, which can be reported at runtime. A *memory transfer bug* happens when not all necessary data was transferred to the device and causes uninitialized memory accesses. We assign the category *memory bug* to questions where a memory error happened, but the cause was unclear from the post. We think that formal methods could help detect these bugs or possibly ensure programmers that such bugs are not present in their program. For instance, CUDA-MEMCHECK [1] and ESBMC-GPU [28] are tools that can detect these kinds of bugs.

Next we consider the memory *performance* sub-theme: ‘manage memory spaces’ and ‘memory performance’. A GPU has to *manage* its own (faster) shared *memory space*. This management can be difficult and error-prone to do but is

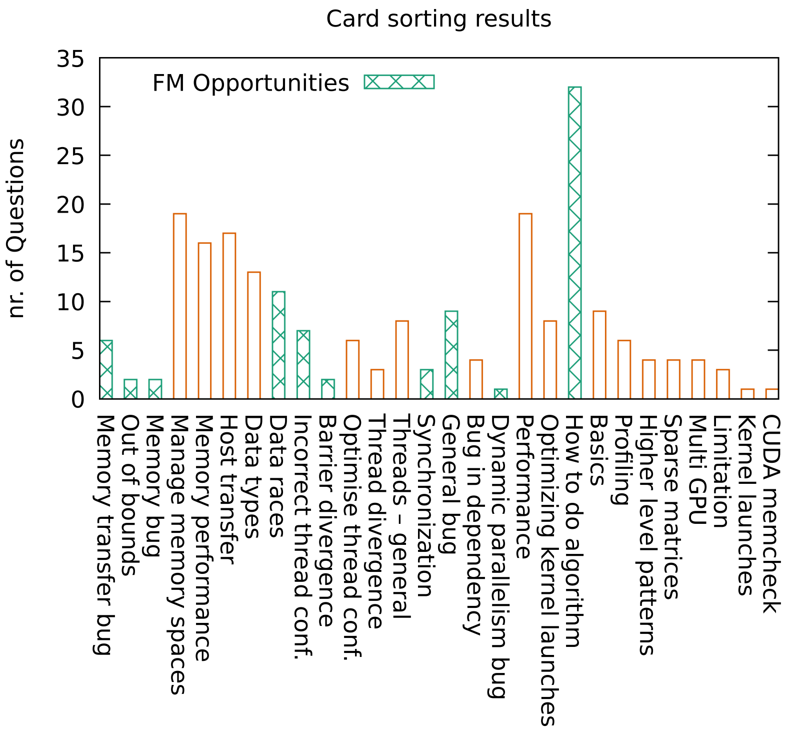

Fig. 1. Results of open card sorting 376 GPGPU related questions. We only show the 220 questions and categories relevant to GPGPU programming. The categories labelled *FM Opportunities* are the ones where we think formal methods could play a role in solving the underlying correctness issues.

an essential optimization strategy. We also added questions related to a better understanding of the memory model here. We label other questions as *memory performance* when they are related to access patterns (coalesced) or other ways to optimize memory usage.

The last two categories are 'host transfer' and 'data types'. Both are related to getting memory from the host to the device. The *host transfer* category is more general. It is related to doing transfers efficiently, asynchronously, transferring the data back, transferring arrays, parameters or constants, and handling arrays too big for global memory. We also assign questions related to aligning and pinning memory here. Actual bugs related to this we report in the 'memory transfer bug' category. We assign questions about overlapping transfers to the 'optimizing kernel launches' category. The *data types* category is more specific. It contains questions related to correctly transferring a composite data type ('struct' in C) and making sure it has a correct corresponding data type on the device. We also consider questions related to Structure of Arrays (SoA) or Arrays of Structures (AoS) here. Although we think that tools can help to

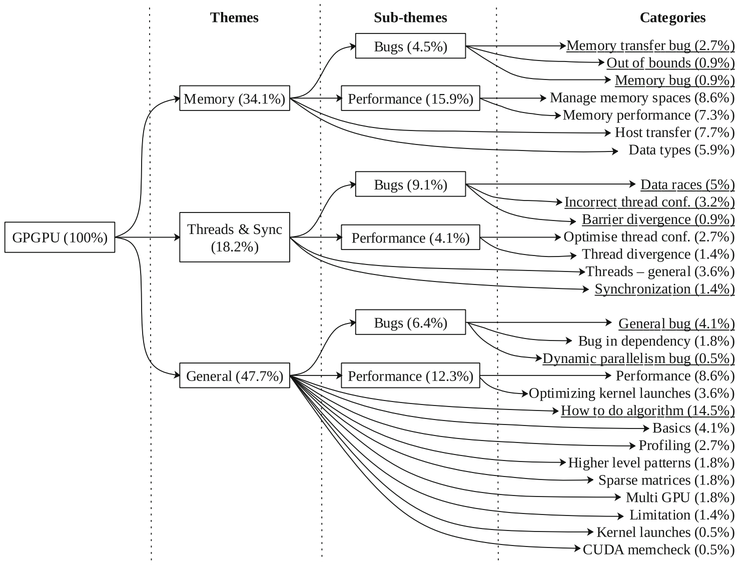

Fig. 2. Overview of the card sort, where we place the categories under themes and sub-themes. Similar to Fig. 1 we only show categories relevant to GPGPU programming. The underlined questions are the ones where we think formal methods could play a role in solving the underlying correctness issues. The percentages indicate how many questions are under a specific category, where 100% corresponds to all 220 relevant GPGPU questions.

solve problems in checking correct correspondence of *data types*, a programming language could do this automatically.

Threads and Synchronization. Under the *bug* sub-theme, we consider 'data races', 'incorrect thread configuration' and 'barrier divergence'. We assign the category *data race* to questions where this occurs. A data race is a problem that is hard to detect: it is non-deterministic by nature, and it is hard to reason about. *Incorrect thread configuration* happens when a programmer configures the wrong number of threads or goes over the maximum amount of threads possible. Some incorrect configurations will be reported at runtime, while others will run without errors but do not process all the input. We assign *barrier divergence* to questions, where not all threads in a thread block reach the same barrier. This is not allowed in the general GPU programming model and leads to undefined results. Data races and barrier divergence bugs are already the study of many formal method tools, like GPUVerify [11] and GKLEE [24]. We think formal methods can also reason about thread configurations, where a tool figures out if the indexing of the input by the threads corresponds with the size of the input or detects memory-related bugs which are caused by incorrect configurations. Another idea is to check whether kernels work the same for each thread configuration.

The 'optimise thread configuration' and 'threads divergence' categories are related to the *performance* sub-theme. When *optimising the amount of threads*, one can choose the number of threads per thread block and how much work each thread does, which both influence performance. *Thread divergence*, on the other hand, could lead to bad performance, which a programmer can sometimes avoid.

The *threads - general* category consists of questions related to understanding the execution model of threads, and what correct configurations are. *Synchronization* is used to prevent data races from happening by using barriers or atomics. General questions on how to use this, about using warp primitives or what can and cannot be synchronized we give this tag. We think formal methods can help people understand when barriers are necessary, or maybe even place barriers automatically. For instance, the Simulee [39] tool can detect unnecessary barriers.

General. First we consider the *bug* sub-theme. We have a *general bug* category, which means something is wrong in the program, but not one of the previously mentioned bugs. This can be incorrect usage of the available resources (e.g. registers), people computing something incorrectly, incorrect use of the Thrust library or it is not yet clear what is wrong. Formal methods, for instance VerCors [12], can check for functional correctness of programs when something is incorrectly calculated. *Bug in dependency* consists of all bugs in Thrust that were fixed in later versions of the library, and are therefore not considered for formal methods later on. *Dynamic parallelism bug* consists of a single question (so/19527391), where a bug was encountered using dynamic parallelism, although it is unclear what exactly went wrong. Formal methods tools could also reason about correctness in this case, although dynamic parallelism would have to be supported.

General *performance* are questions, where people want to understand, given a program, algorithm or function, the performance and how to improve it. Questions about overlapping computations and memory transfers, and ideal scheduling of computation kernels we place in the *optimizing kernel launches* category.

We came across many questions where people wondered how a specific problem or algorithm should be programmed on the GPU, or if a library contained a specific functionality. We placed these in the *how to do algorithm* category. Formal methods could help to prove the equivalence between a sequential and parallel implementation.

The *basics* category has questions related to how certain concepts are called (e.g. what is a thread block), how they work, how specific API calls work, or some basic easy to fix mistakes that prevent correct compilation. Some questions arose about using *higher level patterns* in CUDA and OpenCL, for instance using templated functions. We think these problems are best solved by a beginners GPU programming book or by using a higher-level programming language.

Profiling are questions related to how to use the profiling tools available or how to measure runtimes correctly. *Sparse matrices* are questions on how to process matrices, or on how to use the cuSparse library. *Multi GPU* are questions related to how to use multiple GPUs for computations. The *limitation* category consists of questions related to the limitation of the CUDA/ OpenCL programming model. For example, the CUDA runtime library can only be called from the main scope of a C++ program (so/55819758.) *Kernel launches* are questions related how to start a computation on the GPU correctly. *CUDA memcheck* is about using that specific tool for debugging.

3.3 Insights

Summarizing, we observe that 32.3% of the relevant questions are related to performance, 34.1% to memory, 20% to bugs and 18.2% to threads and synchronization. These are the areas developers on Stack Overflow are most interested in. Performance makes sense since programmers will use a GPU to get better performance, otherwise they would have used the CPU. Memory related questions are important since memory management works quite differently from CPU programs. The transferring of data is error-prone, and the management of memory without race conditions is hard to do. We also think that many developers are just interested in the result: have a faster parallel version of their original (sequential) code, which is related to our 'how to do algorithm' category. Concluding, there is potential for formal methods to help solve correctness related issues that GPGPU programmers experience. We will further discuss this in Sect. 4.

3.4 Threats to Validity

External Validity. There is a bias in the results since we look only at questions located at Stack Overflow. This may not address the general population of GPGPU developers. We suspect that there will be more questions by beginning GPGPU programmers, than by more experienced ones. Therefore, we might not discover the problems of more experienced users.

Internal Validity. As the categories have been manually created, there is an internal bias, meaning that if other people were to perform this study with the same questions, there could be a different outcome. We think that although the categories might be different, the general topics would be similar. Also, part of the categorizing is done together with another PhD student for exactly this reason.

4 Formal Verification Solutions

In Sect. 3, we looked at problems that programmers struggle with when coding in CUDA and OpenCL. In this section we focus on the problems where we think formal methods can make a direct contribution, and provide an overview of tools that (partially) solve these problems. Again, we focus mainly on correctness. First we explain how we selected these verification tools (Sect. 4.1). Next, we discuss for each of the selected problems the available solutions and possible research directions (Sect. 4.2).

4.1 Approach

In order to find as many tools as possible that target the verification of GPU applications, we took the following steps in finding them. First, we looked at the industry. We considered the websites of Nvidia, AMD (gpuopen.com), the Khronos group, and a list[4] found on the IWOCL conference site. Next we looked at important conferences, based on the Microsoft Academic's field ratings.[5] We looked in the areas of programming languages, software verification and parallel computing and selected the following conferences: PLDI, POPL, TACAS, CAV and IPDPS. For each these conferences, we looked through the years 2015–2020.

This was the initial set of tools we considered, and we snowballed, by looking at any tools that the original papers referenced. Lastly, we searched Google Scholar with the following query: "`(cuda OR opencl OR gpu) AND (bugs OR problems OR verification OR formal)`".

4.2 Available Solutions

In this section we consider the problems that we discussed in Sect. 3, where we identified categories. In Table 1, we provide an overview of the tools we found. We distinguish between three types of tools (inspired by Donaldson et al. [15, Chap. 1]): *Dynamic* tools check for one specific input. *Symbolic* tools execute the input symbolically, allowing for more different paths to be tested at once. *Static* tools make (sound) approximations of the source code and will try to prove existence or absence of bugs. We indicate if a tool checks for data races (*Race*), barrier divergence (*Bar*), memory problems (*Mem*), functional correctness (*Func*) or equivalence (*Eq*), or if it helps with synchronization (*Sync*) or thread configuration (*Thr*) in the 'Solves' column. With 'Auto', we refer to the degree of automation: is it completely automatic, or does the user need to be involved in the checking, for instance by writing annotations. The *Corr.* column indicates if the tool can prove the absence of bugs in certain settings. We also list any limitations or other remarks in the table.

Data Races. Ideally, a tool in this category reports existing data races with precise details or guarantees that data races are not present.

Many dynamic tools are practical and require no manual input, but do not guarantee the absence of data races. Solely for checking for a specific input, we think CURD is most suitable, it checks on instruction level, thus can also be used for higher-level languages. Only the approach of Leung et al. [22] gives some guarantees for a dynamic tool and can be used to 'prove' absence of data races for specific invariant kernels. One can combine this approach with other dynamic tools.

Symbolic tools, such as GKLEE and ESBMC-GPU, can test for more input, and one can (mostly) use them automatically although they can also suffer from

[4] https://www.iwocl.org/resources/opencl-libraries-and-toolkits/.
[5] https://academic.microsoft.com/home.

Table 1. Overview of different tools we discuss in this section. We indicate the type of tool, the problems (which we consider in this section) they solve, the degree of automation (Auto.), any correctness guarantees (Corr.) it can give, on which languages it works, and any limitations and other remarks.

Tool	Type	Solves	Auto.	Corr.	Languages	Limitations	Remarks
Oclgrind [31]	Dynamic	Race Bar Mem	High	×	SPIR		Simulates execution
CUDA Memcheck [1]	Dynamic	Race Bar Mem	High	×	CUDA	No global memory for Race	
GRACE [41]	Dynamic	Race	High	×	CUDA	No global memory and atomics for Race	
LDetector [26]	Dynamic	Race	High	×	CUDA	No atomics and intra-warp checks for Race	Does value checking, to determine a race, which might miss races
HAccRG [17]	Hardware	Race	-	×	CUDA		Needs a hardware implementation, but is now simply simulated. Can check fine-grained synchronization
BARRACUDA [16]	Dynamic	Race Bar	High	×	PTX	No intra-warp checks for Race	Can check fine-grained synchronization
CURD [29]	Dynamic	Race Bar	High	×	PTX		Faster version of BARRACUDA
Leung et al. [22]	Dynamic /Static	Race	-	±	CUDA	No atomics	Checks on races for one input and determines if memory accesses are the same for each input. If they are the same, this proves race freedom for all inputs
ARCHER [8]	Dynamic	Race	Medium	×	OpenMP		Runs dynamically on the CPU, not GPU specific
PUG [23]	Static	Race Bar Func	Low	✓	CUDA	Can't check floating point values for Func. Only asserts for Func. Only checks kernels	Scales badly for correctness checking. Can be unsound and incomplete. Needs annotations
GPUVerify [11]	Static	Race Bar Func	Low	✓	CUDA OpenCL	No indirect accesses Only asserts for Func. Only checks kernels	Needs annotations to deal with false positives for races. Is only sound for some CUDA features
VerCors [12]	Static	Race Bar Func	Low	✓	OpenCL	No support for floats (yet)	Needs annotations to prove correctness
GKLEE [24]	Symbolic	Race Bar Func	Medium	×	CUDA (LLVM-IR)	Can't check floating point values for Func. Only asserts for Func.	Difficult to scale for more threads. Generate's concrete tests for races
KLEE-CL [14]	Symbolic	Race Eq	Medium	×	OpenCL		Checks for equivalence on symbolic output, although false positives are possible for this
SESA [25]	Symbolic	Race	High	±	CUDA (LLVM-IR		Similar to GKLEE, but concretize values when possible to reduce runtimes. Can be sound and complete under specific circumstances
ESBMC-GPU [28]	Symbolic	Race Mem Func	High Medium	×	CUDA	Only asserts for Func.	Can be run automatically, but needs assertions for functional correctness checks
Xing et al. [40]	Static	Race	High	±	PTX		Can check fine-grained synchronization. It has to unrolls loops, which can cause unsoundness
Banerjee et al. [10]	Static	Race Eq	High	✓	OpenMP	Equivalent version should be similar	Equivalence checking is sound, but might not be possible for complex programs
WEFT [35]	Static	Race Bar	High	✓	PTX (CUDA)	No global memory and atomics for Race	It is based on a warp specialized programming model. It can only verify programs which are completely predictable, e.g. it cannot have dependencies on the input for memory locations and control flow. It will check named barriers, which are only accesible via PTX
CIVL [36]	Symbolic	Race Eq Mem	Medium	?	OpenMP CUDA Chapel	No atomics	Can use the languages interchangeably, but has no support for specific GPU capabilities. Need some annotations for checking
Alur et al. [7]	Symbolic	Thr	High	±	LLVM-IR (CUDA)		Can only prove block size independence for synchronization free programs
Simulee [39]	Dynamic	DR Bar Sync	High	×	LLVM-IR (CUDA)		Simulates a GPU memory model, and generates memory via evoluationary computing for it
Vericuda [21]	Static	Func	Low	✓	CUDA	Race-free	Needs annotations to prove correctness and can only prove this for race-free programs

longer verification times. GPUVerify is the most practical static verifier, although it needs annotations to overcome false positives. The tool from Xing et al. [40] is interesting and checks on instruction level, but uses loop unrolling, which makes it unsound. It could use ideas from GPUVerify, which generates loop invariants. VerCors can give the most guarantees but needs a serious effort in annotating. For example, see the work of Safari et al. [33], which verifies a prefix-sum algorithm.

WEFT, CIVL, Archer, and the tool of Banerjee et al. [10] serve a more specific purpose, like checking OpenMP or warp-specialised programs.

Overall, many steps have been made to verify data races in GPGPU programs. Checking on instruction level is a good idea since other programming languages benefit from this as well. We also think there are proper steps made to check for fine-grained synchronisation and memory fences which one need for this kind of synchronisation (e.g., BARRACUDA checks on this). From the benchmarks that the authors of the tools consider, it seems to be clear though that there is no tool that always detects or proves the absence of data races. Also, each author uses a different set of benchmarks. It would be interesting to test all the mentioned tools with the benchmark suite created by the work of Schmitz et al. [34], for a fair comparison between tools.

Memory Bugs. Here we look for solutions for the categories: ‘memory bug’, ‘out of bounds’ and ‘memory transfer bug’. Thus, tools should check that memory addresses which are accessed are valid and initialised.

CUDA-MEMCHECK detects the above errors dynamically for CUDA. The OCLgrind tool does the same for OpenCL. ESBMC-GPU verifies on index out of bounds. CIVL checks on array index-out-of bounds. These tools can also check for memory leaks.

For these memory issues, we see an opportunity to check on the instruction level. The dynamic tools seem to cover the properties of interests, but this is not yet the case for the (symbolic) verification tools. For instance, it is unclear if ESBMC-GPU checks on accessing uninitialised memory. Lastly, only VerCors could guarantee correctness for the ‘out of bounds’ issues, but it will only check kernels, not host code and needs annotations.

Barriers and Synchronization. Barrier divergence is also a source of bugs, which can be verified by GPUVerify and GKLEE. CUDA-MEMCHECK detects this dynamically. Another interesting topic, which can help developers with ‘synchronisation’, is placing barriers automatically or notifying the user about unnecessary barriers. The Simulee tool checks for the latter, but no tool addressed the former to the best of our knowledge. Automatic barrier placement could be implemented together with race check tools to afterwards verify for race freedom.

Thread Configuration. The tool by Alur et al. [7] can verify if a synchronisation-free program is blocksize independent: does the program behave the same if the number of blocks is changed, but the total amount of threads stays the same. We think such an approach can be helpful for newer programmers. (And would be a good programming style to begin with.) By making one's program work for any block size, it is easier to optimise. Or even better, verify that one's program behaves the same for any number of threads[6]. A thread-invariant program lets one freely try different thread configurations without introducing new bugs. Thus, we see an opportunity for verification tools addressing this.

Dynamic Parallelism. As far as we know, there are no tools that support dynamic parallelism, although we are not sure if tools working at the instruction level, e.g. BARRACUDA, support this. Support for dynamic parallelism is the first step to ensure that a tool can check kernels using this concept. One can also come across new bugs like data races between parent and child kernels. Specific to dynamic parallelism is the fact that there is a maximum recursion depth of new kernels and a maximum number of child kernels. A formal methods tool can check both of these restrictions.

Functional Correctness. VerCors [13] allows deductive checking of functional correctness of programs, although it needs non-trivial annotations.

On a similar vein, the work of Kojima et al. [20] proposes a Hoare logic for GPU programs, which the Vericuda tool [21] verifies when one provides Hoare tuples. However, the latter tool requires that the checked program is data race free, which should be verified by another program.

ESBMC-GPU, CIVL, GPUVerify and GKLEE allow the programmer to place assertions. These assertions do not give complete correctness but allow more flexibility in checking certain aspects of the program.

We think VerCors has potential, although the need for annotations makes it difficult to use out of the box. An interesting research direction is making the reuse of annotations easier after a program has been slightly changed, e.g. due to an optimisation.

Equivalence Checking. Instead of fully verifying a specification, one can do equivalence checking: take a (simple), possibly sequential, version of a program, which you know is correct and prove that a parallel implementation is equivalent. The CIVL tool can do this. Kamil et al. [19] use a similar approach. They transform Fortran stencil codes to Halide (an image processing DSL), and proof functional equality, while being able to optimise the program in Halide further. The tool by Banerjee et al. [10] is similar. It verifies equivalence for parallelising loop transformations from OpenMP and also verifies data race freedom.

[6] https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

4.3 Research Directions

We think much progress has already been made by formal methods that address many issues that developers encounter. We make the following observations.

In general, we think that checking on instruction level is valuable. Typically, all GPU programs will eventually compile to the instruction level, and thus allows the tool to be used for more programming languages.

No verification tool is completely sound yet, which might be impossible for the full flexibility of the CUDA and OpenCL languages, but should be the goal. Tools should support as many program features as possible while staying sound. Certainly, since programmers use a lot of low-level features when optimising code, this is an ambitious goal.

Most verification tools only check the GPU kernels, but the host code must also be correct. Bugs related to memory initialization, memory transfers and thread configuration are often made on the host side, can be hard to spot and should be relatively easy to tackle with verification tools.

Another take on this is to identify which patterns and programming features are sound to verify. This can give rise to a particular programming style, which can be enforced by a different (domain-specific) language.

In the papers presenting the various tools, those tools are compared with each other to show that for specific kernels, the new tool is, at that point in time, the best. It would be better to use a standard benchmark suite, like the suite by Schmitz et al. [34], which is uniformly used and addresses the errors we mention in this paper. Additionally, it should support all the CUDA and OpenCL features. This suite then makes it clear what errors tools can check and what programming features they do or do not support. For instance, we think that tools that deal with fine-grained synchronisation are essential.

5 Related Work

GPGPU Problems. The study by Wu et al. [39] is similar to our work. Instead of Stack Overflow, they look at open source repositories on Github to collect CUDA bugs. They identify 5 root causes for bugs, which is coarser than our results. We can match most of our categories with one of their root causes. Only their 'poor portability' we can not match, and is more related to specific platforms issues, which were questions we marked as irrelevant. Also, the nature of Stack Overflow means we have more questions related to solely understanding GPU programming (e.g. 'Basics' or 'How to do algorithm') and are not things you could find in commit messages. Because of that reason, the exact numbers on how often certain issues arise are hard to compare, but we don't think that is too important. Both of these methods give a good overview of what kind of bugs to expect whilst GPGPU programming.

The work of Donaldson et al. [9, Chap. 1] gives an overview of what kind of correctness issues occur with GPGPU programming and gives a comparison between the tools GPUVerify, GKLEE, Oclgrind and CUDA-MEMCHECK.

They name four different correctness issues: *data races*, *weak memory behaviours*, *lack of forward progress guarantees* and *floating point accuracy*. Of these issues, we have only come across *data races* in our study. We think the other issues are more particular for experienced users, and less so for novice users. As mentioned before, we think Stack Overflow attracts mostly novice users. The taxonomy made by Donaldson et al. of the considered tools inspired the current work, although we consider a wider range of tools overall.

Stack Overflow Studies. There were many other studies performed on Stack Overflow concerning other subjects, for example concurrency [5,30] mobile development [32] and machine learning [18]. In [5,30,32] topic modelling is used to categorize all the questions. We chose to not use topic modeling, since we think that we can make a finer subdivision of the categories with open card sorting. In [18] something more related to our work was done, but experts pre-determined the categories. In our case the goal was to discover problems, therefore it makes no sense to pre-determine the categories.

6 Discussion

In this work, we showed the problems GPGPU programmers struggle with, while programming for the GPU using OpenCL or CUDA. We see that memory, synchronization, threads and performance are essential topics for GPGPU programming. Next, we looked at (formal method) tools and how they address the correctness issues we found. In general, the research community addresses most problems, but we identified several interesting research directions.

Acknowledgements and Data Availibility Statement. We want to thank Jan Martens for his help with the card sorting.

The data used for the categorization with card sorting is available in the Figshare repository: https://doi.org/10.4121/12988781 [38].

References

1. CUDA-MEMCHECK, June 2020. https://docs.nvidia.com/cuda/cuda-memcheck
2. CUDA Programming Guide, July 2020. http://docs.nvidia.com/cuda/cuda-c-programming-guide/
3. Parallel Thread Execution ISA Version 7.0, July 2020. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html
4. SPIR - The Industry Open Standard Intermediate Language for Parallel Compute and Graphics, July 2020. https://www.khronos.org/spir/
5. Ahmed, S., Bagherzadeh, M.: What do concurrency developers ask about? a large-scale study using stack overflow. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, pp. 1–10. Association for Computing Machinery, Oulu, October 2018. https://doi.org/10.1145/3239235.3239524

6. Alglave, J., et al.: GPU concurrency: weak behaviours and programming assumptions. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 2015, pp. 577–591. ACM Press, Istanbul (2015). https://doi.org/10.1145/2694344.2694391
7. Alur, R., Devietti, J., Singhania, N.: Block-size independence for GPU programs. In: Podelski, A. (ed.) SAS 2018. LNCS, vol. 11002, pp. 107–126. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_9
8. Atzeni, S., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62. IEEE, Chicago, May 2016. https://doi.org/10.1109/IPDPS.2016.68
9. Azad, H.S.: Advances in GPU Research and Practice, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2016)
10. Banerjee, K., Banerjee, S., Sarkar, S.: Data-race detection: the missing piece for an end-to-end semantic equivalence checker for parallelizing transformations of array-intensive programs. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY 2016, pp. 1–8. Association for Computing Machinery, Santa Barbara, June 2016. https://doi.org/10.1145/2935323.2935324
11. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 113–132. ACM, New York (2012). https://doi.org/10.1145/2384616.2384625
12. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
13. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014). https://doi.org/10.1016/j.scico.2014.03.013
14. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 203–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_18
15. Donaldson, A.F., Ketema, J., Sorensen, T., Wickerson, J.: Forward progress on GPU concurrency (invited talk). In: Meyer, R., Nestmann, U. (eds.) 28th International Conference on Concurrency Theory (CONCUR 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 85, pp. 1:1–1:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2017). https://doi.org/10.4230/LIPIcs.CONCUR.2017.1
16. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: binary-level analysis of runtime RAces in CUDA programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 126–140. Association for Computing Machinery, Barcelona, June 2017. https://doi.org/10.1145/3062341.3062342
17. Holey, A., Mekkat, V., Zhai, A.: HAccRG: hardware-accelerated data race detection in GPUs. In: 2013 42nd International Conference on Parallel Processing, pp. 60–69. IEEE, Lyon, October 2013. https://doi.org/10.1109/ICPP.2013.15
18. Islam, M.J., Nguyen, H.A., Pan, R., Rajan, H.: What do developers ask about ML libraries? A large-scale study using stack overflow. ArXiv: 1906.11940 (Cs), June 2019

19. Kamil, S., Cheung, A., Itzhaky, S., Solar-Lezama, A.: Verified lifting of stencil computations. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 711–726. Association for Computing Machinery, Santa Barbara, June 2016. https://doi.org/10.1145/2908080.2908117
20. Kojima, K., Igarashi, A.: A hoare logic for GPU kernels. ACM Trans. Comput. Log. **18**(1), 3:1–3:43 (2017). https://doi.org/10.1145/3001834
21. Kojima, K., Imanishi, A., Igarashi, A.: Automated verification of functional correctness of race-free GPU programs. J. Autom. Reason. **60**(3), 279–298 (2018). https://doi.org/10.1007/s10817-017-9428-2
22. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. SIGPLAN Not. **47**(6), 383–394 (2012). https://doi.org/10.1145/2345156.2254110
23. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2010, p. 187. ACM Press, Santa Fe (2010). https://doi.org/10.1145/1882291.1882320
24. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 215–224. ACM, New York (2012). https://doi.org/10.1145/2145816.2145844
25. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: SC 2014: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 179–190, November 2014. https://doi.org/10.1109/SC.2014.20
26. Li, P., et al.: LD: low-overhead GPU race detection without access monitoring. ACM Trans. Archit. Code Optim. **14**(1), 1–25 (2017). https://doi.org/10.1145/3046678
27. Menzies, T., Williams, L., Zimmermann, T.: Perspectives on Data Science for Software Engineering, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2016)
28. Monteiro, F.R., et al.: ESBMC-GPU a context-bounded model checking tool to verify CUDA programs. Sci. Comput. Program. **152**, 63–69 (2018). https://doi.org/10.1016/j.scico.2017.09.005
29. Peng, Y., Grover, V., Devietti, J.: CURD: a dynamic CUDA race detector. In: PLDI 2018, pp. 390–403. Association for Computing Machinery, Philadelphia, June 2018. https://doi.org/10.1145/3192366.3192368
30. Pinto, G., Torres, W., Castor, F.: A study on the most popular questions about concurrent programming. In: Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools - PLATEAU 2015, pp. 39–46. ACM Press, Pittsburgh (2015). https://doi.org/10.1145/2846680.2846687
31. Price, J., McIntosh-Smith, S.: Oclgrind: an extensible OpenCL device simulator. In: Proceedings of the 3rd International Workshop on OpenCL - IWOCL 2015, pp. 1–7. ACM Press, Palo Alto (2015). https://doi.org/10.1145/2791321.2791333
32. Rosen, C., Shihab, E.: What are mobile developers asking about? A large scale study using stack overflow. Empir. Softw. Eng. **21**(3), 1192–1223 (2016). https://doi.org/10.1007/s10664-015-9379-3
33. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A. (eds.) NFM 2020. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10

34. Schmitz, A., Protze, J., Yu, L., Schwitanski, S., Müller, M.S.: DataRaceOnAccelerator – a micro-benchmark suite for evaluating correctness tools targeting accelerators. In: Schwardmann, U., et al. (eds.) Euro-Par 2019. LNCS, vol. 11997, pp. 245–257. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48340-1_19
35. Sharma, R., Bauer, M., Aiken, A.: Verification of producer-consumer synchronization in GPU programs. In: PLDI 2015, pp. 88–98. Association for Computing Machinery, Portland, June 2015. https://doi.org/10.1145/2737924.2737962
36. Siegel, S.F., et al.: CIVL: the concurrency intermediate verification language. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 2015, pp. 1–12. ACM Press, Austin (2015). https://doi.org/10.1145/2807591.2807635
37. Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G., Rakamaric, Z.: Portable inter-workgroup barrier synchronisation for GPUs. In: OOPSLA 2016, p. 20 (2016). https://doi.org/10.1145/3022671.2984032
38. van den Haak, L.B., Wijs, A., van den Brand, M., Huisman, M.: Card sorting data for Formal methods for GPGPU programming: is the demand met?, September 2020. https://doi.org/10.4121/12988781
39. Wu, M., Zhou, H., Zhang, L., Liu, C., Zhang, Y.: Characterizing and detecting CUDA program bugs. ArXiv: 1905.01833 (Cs), May 2019
40. Xing, Y., Huang, B.Y., Gupta, A., Malik, S.: A formal instruction-level GPU model for scalable verification. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8, November 2018. https://doi.org/10.1145/3240765.3240771
41. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: a low-overhead mechanism for detecting data races in GPU programs. SIGPLAN Not. **46**(8), 135–146 (2011). https://doi.org/10.1145/2038037.1941574