International Journal of Parallel Programming (2021) 49:761–775
https://doi.org/10.1007/s10766-021-00703-4

Bounds Checking on GPU

Troels Henriksen[1]

Received: 27 October 2020 / Accepted: 2 March 2021 / Published online: 25 March 2021

Abstract
We present a simple compilation strategy for safety-checking array indexing in high-level languages on GPUs. Our technique does not depend on hardware support for abnormal termination, and is designed to be efficient in the non-failing case. We rely on certain properties of array languages, namely the absence of arbitrary cross-thread communication, to ensure well-defined execution in the presence of failures. We have implemented our technique in the compiler for the functional array language Futhark, and an empirical evaluation on 19 benchmarks shows that the geometric mean overhead of checking array indexes is respectively 4% and 6% on two different GPUs.

Keywords GPU · Functional programming · Compilers

1 Introduction

Programming languages can be divided roughly into two categories: *unsafe* languages, where programming errors can lead to unpredictable results at run-time; and *safe* languages, where all risky operations are guarded by run-time checks. Consider array indexing, where an invalid index will lead an unsafe language to read from an invalid memory address. At best, the operating system will stop the program, but at worst, the program will silently produce invalid results. A safe language will perform *bounds checking* to verify that the array index is within the bounds of the array, and if not, signal that something is amiss. Some languages perform an *abnormal termination* of the program and print an error message pointing to the offending program statement. Other languages throw an exception, allowing the problem to be handled by the program itself. The crucial property is that the resulting behaviour is well-defined. We use array indexing as the motivating example, but we are concerned with all safety checks that can be condensed to a single boolean expression; for example integer division by zero.

✉ Troels Henriksen
athas@sigkill.dk

[1] University of Copenhagen, Copenhagen, Denmark

Users of unsafe languages are often wary of the run-time overhead of performing safety checks. However, it has been known since even the early days of high-level languages that bounds errors are easy to make and can have disastrous consequences [15], and hence most languages provide at least the option of automatically checking risky operations. In this paper we distinguish *failures* from *errors*. A *failure* is a checked operation that fails in some controlled manner. For example an array index that is discovered to be out-of-bounds. An *error* is a misuse of some low-level API or language construct that causes undefined behaviour. For example, writing to an invalid address. A safe programming language must ensure that anything that would be an error will instead become a failure.[1]

GPUs have long been popular for general-purpose parallel programming, and several high-level languages support compilation to GPU, including Accelerate [5], Lift [19], Julia [4], ×10 [7], Harlan [16], APL [12,17], and SaC [11]. As these are all high-level languages, most of them provide at least the option of performing bounds checking when running on a CPU, but none of them can perform bounds checking in generated GPU code. One important reason is that the most popular GPGPU APIs (OpenCL and CUDA) do not provide good support for abnormal termination of a running GPU kernel.

For example, CUDA provides an `assert()` macro that, if it fails, will terminate the calling kernel. However, it will also invalidate the entire CUDA driver context, meaning that memory that has been copied to the GPU by the current process becomes unavailable. Further, the error message will be printed to the standard error stream, which may be difficult to capture and propagate (e.g. by throwing an exception on the CPU). This means failures cannot be handled in any way other than completely scrubbing the entire GPU state, including even data that was not available to the failing kernel, and restarting from scratch, which is often not acceptable. While a single GPU thread can always terminate itself, this can introduce deadlocks (see Sect. 2.3), which is an error. OpenCL is similar, except there is *no* way for a GPU thread to abnormally terminate an entire running kernel.

Functional array languages [3], where programs consist primarily of bulk operations such as `map`, `reduce`, and rank-polymorphic "vectorised" operators, do not contain indexing errors, as such operations are guaranteed to be in-bounds. However, some algorithms do still require ad hoc indexing, in particular when we use arrays to encode more complicated structures, such as graphs. In particular, parallel "gather" and "scatter" operations have all the same risks as traditional scalar array indexing, and should therefore be checked at run-time. Fortunately, as we shall see, array languages based on bulk operations have certain properties that enable a particularly efficient implementation of run-time safety checks.

The contribution of this paper is a compilation strategy for inserting safety checks in GPU code generated by compilers for high-level parallel languages, without relying on support for abnormal termination or error reporting in the GPU API or hardware itself. Our design goals are the following:

[1] The error/failure nomenclature is more or less arbitrary and not standard or common, but the distinction is important for this paper. In C, the term *undefined behaviour* is a close (but not exact) analogue to what we call *errors*.

Completeness all possible safety checks that are expressible as a boolean expression can be handled.

Efficiency overhead must be low, as programmers are quick to turn off safety checks that they believe are detrimental to performance. However, we focus only on performance of the *non-failing* case, as we assume failures are rare and exceptional situations.

Robustness the GPU driver context must remain operational even in the presence of safety check failures—*errors* must not occur, as far as the GPU programming API is concerned.

Quality of reporting we must be able to produce accurate information about the source of the failure, phrased in terms of original high-level language (e.g. the failing expression and index) rather than low-level details (e.g. the invalid address).

Note that we are not claiming to safety-check GPU kernels hand-written in low-level languages such as CUDA and OpenCL. Our strategy depends on properties that are straightforward to guarantee in code generated by compilers for deterministic parallel programming languages, but which would not hold for languages that support programmer-written low-level communication between threads.

1.1 Prior Work

Several tools for detecting invalid memory accesses have been implemented for GPUs. Oclgrind [18] presents itself as an OpenCL platform which runs all kernels in an interpreter and reports accesses to invalid memory locations. However, Oclgrind is primarily a debugging tool, as it runs far slower than real hardware. NVIDIA provides the similar tool `cuda-memcheck`, which detects invalid memory accesses, but as the instrumented kernels run with a significant run-time overhead, it is also a debugging tool, and not intended for code running in production.

A more efficient (and less precise) tool is the vendor-agnostic clARMOR [9], which surrounds every allocation with an area of unique *canary values*. If at any point any of these values have been changed, then it must be because of an out-of-bounds write. The overhead is minor (10% on average), but clARMOR can detect only invalid writes, not reads, and cannot identify exactly *when* the invalid access occurred.

All these tools are low-level and concerned with the semantics of OpenCL or CUDA kernel code, and so are not suitable for implementing bounds checking for a high-level language. In particular, they would not be able to live up to our expectations for error messages. Further, all such tools run the risk of *false negatives*, where a bounds failure ends up corrupting memory at an address that is valid, but unintended. This cannot be detected by tools that merely verify addresses.

There is also the option of using entirely static techniques to perform bounds checking, such as dependent types [21], which demand that the programmer provides a proof that all indexing is safe before the type checker accepts the program. No checking is then needed at run-time. However, dependently typed programming languages are still an active research topic with regards to both programming ergonomics and run-time performance, and so are not necessarily a good choice in the near future for performance-oriented languages. In either case, such techniques are complimentary to run-time checking: where the programmer is willing to invest the time to provide a proof of safety, we can turn off run-time checks, while keeping checks in the unverified parts of the program.

A closely related problem is *de-optimisation* in the context of JIT compilation, where an assumption made by the JIT compiler may turn out to be false at run-time, and execution must be rolled back in order run a slower interpreted version of the code. Prior work on JIT compiling R to GPU code [10] uses a technique similar to the approach we will be discussing in Sect. 2, but without the optimisations of Sects. 2.2 and 3, and of course also without error messages.

1.2 Nomenclature and Technicalities

We use OpenCL terminology for GPU concepts. Despite the naming differences with CUDA, the concepts are identical, and our approach works just as well with CUDA as with OpenCL. An OpenCL *work-group* corresponds to what CUDA calls a *thread block*, and is a collection of threads that executes together and may communicate with each other. OpenCL *local memory* corresponds to CUDA *shared memory*.

We are assuming a particularly simple and conventional GPU model, with the GPU operating as a co-processor that merely receives commands and data from the CPU. In particular, we assume a kernel cannot enqueue new kernels, and cannot allocate or free memory. Some real GPUs do have these capabilities, but they have significant performance caveats, are not crucial to GPU programming, and in particular are not used in the code generated by any of the previously mentioned high-level languages.

2 Design and Implementation

As currently popular GPGPU APIs (e.g. OpenCL and CUDA) do not permit abnormal termination of GPU kernels, we need to turn failing executions into normal kernel termination, and somehow communicate the failure back to the CPU. It is important that we do not at any point execute errors, such as reading from invalid memory addresses.

A simple solution is to allocate a single 32-bit integer in GPU memory, which we call `global_failure`, and which we use to track failures. We use the convention that a value of -1 means “no failure”, and other values indicate that a failure has occurred. When a failure occurs, the failing GPU thread writes a non-negative integer to `global_failure` and immediately `returns`, which stops the thread. After every kernel execution, we can then copy the value of `global_failure` back the CPU, and if it contains a non-negative value, propagate the failure using conventional CPU mechanisms, such as throwing an exception or printing an error message. This idea is the foundation of our approach, but in this simple form it has significant problems:

1. It is *uninformative*, because simply knowing that the program failed is not enough to provide a good error message. For an array indexing failure, we usually wish to provide the expression that failed, the attempted index, and the size of the array.

2. It is *slow*, because it requires a global synchronisation after every kernel execution, to verify whether it is safe to execute the next kernel. GPUs perform well when given a large queue of work to process at their own pace, not when they constantly stop to transfer 32 bits back to the CPU for inspection, and have to wait for a go-ahead before proceeding.
3. It is *wrong*, because GPU threads are not isolated, but may communicate through barriers or other synchronisation mechanisms. In particular, it is undefined behaviour for a barrier to be executed by *at least one* but *not all* threads. We cannot in general abort the execution of a single thread without risking errors.

We will now explain at a high level how to address these problems, accompanied by a sketch of a concrete OpenCL implementation.

2.1 Better Failure Information

Treating failure as a boolean state, without revealing the source of the failure, is not very user-friendly. Our solution is to assign each distinct *failure point* in the program a unique number: a *failure code*. A failure point is a program location where a safety check is performed. If the check fails, the corresponding failure code is written to `global_failure`. The write is done with atomic compare-and-swap to ensure that any existing failure code is not clobbered. The distinguished value -1 indicates that no failure has yet occurred. We use compare-and-swap to ensure that at most one thread can change `global_failure` from -1 to a failure code. This implies that if multiple threads contain failures, it is not deterministic which of them get to report it. We can only report a single failure to the user, and multiple runs of the same failing program may produce different error messages. During compilation of the original program, we construct a table that maps failure codes to the original source code locations. When we check `global_failure` at run-time on the CPU, we can then identify the exact expression that gave rise to the failure.

To provide human-readable error messages, we associate each failure point with a `printf()`-style format string such as the following:

```
"index %d out of bounds for array of size %d"
```

For simplicity we assume that `%d` is the only format specifier that can occur, but each distinct format string can contain a different number of format specifiers. We then pre-allocate an `int` array `global_failure_args` in GPU memory that is big enough to hold all parameters for the largest format string. When a thread changes `global_failure`, it also writes to `global_failure_args` the integers corresponding to the format string arguments. When the CPU detects the failure after reading `global_failure`, it uses the failure code to look up the corresponding format string and instantiates it with arguments from `global_failure_args`. Note that the CPU only accesses `global_failure_args` when a failure has occurred, so performance of the non-failing case is not affected.

For a non-recursive language, each failure point can be reached through a finite number of different call paths, and stack traces can be provided by generating a distinct format string for each possible path. The recursive case is more difficult, and outside the scope of this paper, but can possibly be handled by simply deciding on a maximum backtrace length, and then representing a failure point as an entire array of source locations, rather than a single one.

2.2 Asynchronous Failure Checking

It is expensive to check `global_failure` on the CPU after every kernel execution. We should do so only when we are, for other reasons, required to synchronise with the GPU. This is typically whenever we need to copy data from GPU to CPU, such as when making control flow decisions based on GPU results, or when we need the final program result.

Simply delaying the check is not safe, as kernel $i+1$ may contain unchecked operations that are safe if and only if the preceding kernel i completed successfully. To address this, we add a prelude to every GPU kernel body where each thread checks `global_failure`. If `global_failure` contains a failure code, that must mean one of the preceding kernels has encountered a failure, and so the all threads of the current kernel terminate immediately. This is an improvement, because checking `global_failure` on the GPU is much faster than checking it on the CPU, and does not involve any CPU/GPU synchronisation. The overhead is a single easily cached global memory read for every thread, which is in most cases negligible, and Sect. 3.5 shows cases where even this can be elided.

This technique means that an arbitrary (but finite) amount of time can pass from the time that a GPU kernel writes to `global_failure`, to the time that the failure is reported to the CPU. Specifically, the time is bounded in the worst case by the time it would have taken the program to finish successfully. We consider this an easy price to pay in return for improving the performance of the non-failing case.

2.3 Cross-thread Communication

In general, a GPU thread cannot safely terminate its own execution, as other threads may be waiting for it in a barrier. Consider Fig. 1a, which shows a typical OpenCL summation kernel, where each thread performs a sequential summation of a chunk of the input, followed by a parallel summation of the per-thread results within the GPU work-group. This requires a *barrier* (line 25) to ensure that all threads have written their result to the shared `sums` array before the parallel reduction takes place.

In this kernel, the sequential part involves indirect indexing, where the array `js` contains values used to index `vs`. These indexes can be out-of-bounds, which on Fig. 1a is handled by setting `global_failure` and terminating the thread with `return`. But this is risky, as other threads may already be waiting at the barrier, which will never be reached by the failing thread.

The solution, shown on Fig. 1b, identifies the location of the next barrier in the code, places a distinct label just before it (line 24), and then `goto` that label instead of immediately terminating. We call this label/barrier pair a *synchronisation point*. Immediately after the synchronisation point, *all* threads in the work-group check whether any of them have failed, by inspecting `global_failure`, and terminate

```
kernel sum
  (int *global_failure,
   int n, int *js,
   int m, int *vs,
   ...) {
  local int sums[GROUP_SIZE];
  int gtid = get_global_id(0);
  int ltid = get_local_id(0);
  int k = get_global_size(0);
  int acc = 0;
  for (int i = gtid;
       i < n;
       i += k) {
    int j = js[i];
    if (j < 0 || j >= m) {
      *global_failure = 1;
      return;
    }
    acc += vs[j];
  }

  sums[ltid] = acc;


  barrier();





  // Perform parallel
  // reduction of sums...
}
```

(a) Incorrect failure handling.

```
kernel sum
  (int *global_failure,
   int n, int *js,
   int m, int *vs,
   ...) {
  local int sums[GROUP_SIZE];
  int gtid = get_global_id(0);
  int ltid = get_local_id(0);
  int k = get_global_size(0);
  int acc = 0;
  for (int i = gtid;
       i < n;
       i += k) {
    int j = js[i];
    if (j < 0 || j >= m) {
      *global_failure = 1;
      goto sync;
    }
    acc += vs[j];
  }

  sums[ltid] = acc;

  sync:
  barrier();
  if (*global_failure != -1) {
    return;
  }
  barrier();

  // Perform parallel
  // reduction of sums...
}
```

(b) Correct failure handling.

Fig. 1 OpenCL-like pseudo-code for kernel that sums an indirectly indexed array of integers

if so. The barrier implies a memory fence, so the threads within the work-group will have a consistent view of the `global_failure` variable. We also need a barrier immediately after this check, because other failure points may occur in the remainder of the kernel, and these would also set `global_failure`. A kernel may have multiple synchronisation points, each identified with a distinct label. We must ensure that all kernels end with a final synchronisation point, such that there is always a place to jump from a failure point.

Viewed as a control flow graph, the kernel code must have the property that every node that controls a failure point has a postdominator that contains a synchronisation point, and that there are no barriers on the path to the synchronisation point. It is crucial that this is a postdominator, because even non-failing executions must reach it. This property does not hold for arbitrary GPU kernels, but it is straightforward to ensure it when compiling array languages, because all cross-thread communication (and hence, barriers) is implicit in the source program and controlled by the compiler. For example, consider the control flow graph on Fig. 2. If a failure occurs in node **B**, where should we jump? The choice of the next barrier is not decided until node

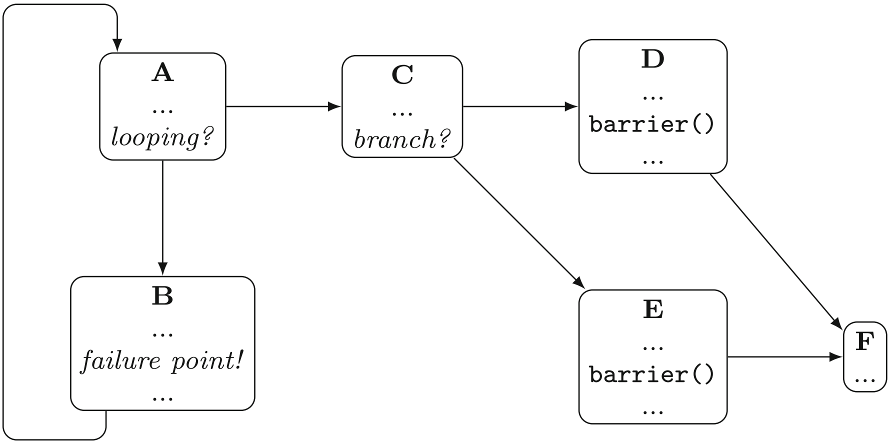


Fig. 2 Control flow graph where the failure point in node **B** cannot know the location of the next barrier

C. The compiler must insert a synchronisation point in **C** to ensure that there is an unambiguous location to which the failure point can jump.

GPU programmers usually view `goto` with scepticism. Apart from the usual problems [8], unrestricted use of `goto` can cause irreducible control flow, which is in general highly inefficient and sometimes unsupported on SIMT architectures. However, our use of `goto` to jump to postdominators does not cause irreducible control flow, and as we shall see in Sect. 4, performance on contemporary GPUs is good.

3 Further Optimisations

This section shows additional optimisations and implementation hints that reduce the overhead of failure checking. The sum impact of these optimisations is shown in Sect. 4.

3.1 Avoiding Global Memory Accesses

Some GPU kernels contain significant cross-thread communication within each work-group, which must be interleaved with checking the `global_failure` variable before every barrier. This can be a bottleneck, as `global_failure` is stored in global memory, and the kernel may otherwise use mostly the much faster local memory. To address this, we introduce a boolean variable `local_failure`, stored in local memory, that indicates whether a failure has occurred within the current work-group. When a thread fails, we set *both* `global_failure` and `local_failure`, but synchronisation points check only the latter. This is sufficient to ensure safety, as GPU work-groups cannot communicate with each other, and hence cannot be affected by a failure in another work-group. The final code emitted for a failure point is shown on Fig. 3c, and the code for a synchronisation point on Fig. 3d

`int *global_failure`
: If pointed-to value is non-negative, the program is in a failing state.

`int failure_is_an_option`
: Whether `*global_failure` is potentially non-negative.

`int *global_failure_args`
: An array of values indicating precisely the failure that has occurred, e.g. the invalid index that was attempted. Never read from a kernel, but only written to.

(a) Parameters added to every kernel. Arguments for `global_failure` and `global_failure_args` are set on the host when the kernel is first created, but `failure_is_an_option` must be set whenever the kernel is enqueued.

```
volatile __local bool local_failure;
if (failure_is_an_option) {
  if (get_local_id(0) == 0) {
    local_failure = *global_failure >= 0;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  if (local_failure) {
    return;
  }
} else {
  local_failure = false;
}
barrier(CLK_LOCAL_MEM_FENCE);
```

(b) The failure handling prelude added to generated OpenCL kernels. See fig. 3a for the meaning of the kernel parameters used.

```
local_failure = true;
if (atomic_cmpxchg(global_failure, -1, k) < 0) {
  // write to global_failure_args...
}
goto sync;
```

(c) The code emitted whenever a failure occurs inside a kernel, where k is a unique non-negative integer identifying the failure, and *sync* a label identifying the next failure synchronisation point (see fig. 3d).

```
sync:
barrier(CLK_LOCAL_MEM_FENCE);
if (local_failure) {
  return;
}
```

(d) Code for a synchronisation point, where *sync* is a distinct label referenced in preceding `goto` statements (see fig. 3c).

Fig. 3 Essential code fragments for our implementation of GPU bounds checking. This lists kernel parameters and code only

3.2 Avoiding Unnecessary Failure Checking

As we discussed in Sect. 2.2, threads in a running kernel initially check `global_failure` for whether a failure has occurred in a previous kernel. This requires multiple barriers to ensure that threads in the current kernel do not run ahead, fail, and set global_failure before all other threads have had a chance to read its initial value. While barriers are relatively cheap, we have observed that this initial checking still has a cost for very simple kernels. In many cases, we have run-time knowledge that no kernels with failure points have been enqueued since the last time we checked global_failure, and hence checking it is wasteful.

We address this by adding to every kernel another int-typed parameter, failure_is_an_option, that indicates whether *global_failure is potentially set. The value for this parameter is provided by the CPU when the kernel is enqueued.

The full prelude added to OpenCL kernels for failure checking is shown on Fig. 3b, and the pertinent kernel parameters on Fig. 3a. Note that we still need a barrier to ensure local_failure is properly initialised.

The failure_is_an_option parameter corresponds to an ordinary variable maintained by the CPU. It is initially zero, and set whenever we enqueue a kernel that contains failure points. Whenever the CPU synchronises with the GPU, and would normally copy global_failure back to the CPU to check its value, we first check failure_is_an_option. If zero, that means there is no reason to check global_failure. Our motivation is avoiding the latency of initiating a transfer, as copying a single 32-bit word of course takes very little bandwidth. After any CPU–GPU synchronisation where global_failure is checked, we reset failure_is_an_option to zero.

3.3 Avoiding Synchronisation Points

Many kernels, particularly those corresponding to a map, contain no communication between threads, and hence no barriers. For these kernels, failure points can simply return.

3.4 Non-failing Kernels

Many kernels contain no failure points, and are guaranteed to execute successfully. These kernels must still check global_failure when they start, because this guarantee may be predicated on the successful execution of previous kernels, but they do not need the failure_is_an_option or global_failure_args parameters, and their kernel prelude can be simplified to the following:

```
if (*global_failure >= 0)  return;
```

3.5 Failure-Tolerant Kernels

Some particularly simple kernels are able to execute safely (i.e. error-free) even when previous kernels have failed, typically because they merely copy or replicate memory, possibly with an index transformation. Matrix transposition is an example of such a kernel. For these kernels we can eliminate all failure checking entirely. This is because failures cannot result in memory becoming *inaccessible*; it can only result in the values stored being *wrong*, and these simple kernels are not sensitive to the values they are copying.

4 Experiments

We have implemented the presented technique in the compiler for Futhark [14], a functional array language that can be compiled to OpenCL and CUDA. Apart from checking array indexes, we also check for integer division by zero, as well as arbitrary programmer-provided assertions. These can be handled using the same approach as bounds checking. Futhark is a purely functional language, and so is not suitable for writing full applications. Instead, a compiled Futhark program presents a C API, with Futhark *entry points* exposed as C functions, which are then called by programs written in other programming languages. Futhark does not support exceptions or similar error handling mechanisms, so in the event of a failure, the error message is simply propagated to the return value of the C API, where it is made available for the caller to do with as they wish. One option is of course to print the message to the console and then terminate the entire process, but the GPU and Futhark state remains intact, including the data that was passed to the Futhark entry point, so it is also possible to continue execution with other data.

To investigate the efficiency of our implementation technique, we have measured the run-time of a range of Futhark programs compiled with and without bounds checking enabled. The full Futhark benchmark suite[2] contains 41 programs ported from Accelerate [5], FinPar [2], Rodinia [6], and Parboil [20]. Of these, 19 benchmarks require bounds checks in GPU kernels, and are the ones we use in our experiments. See Table 1 for a table of the benchmarks and workloads. Since our focus is the relative cost of bounds checking, we do not compare our performance with the original hand-written benchmark implementations. Prior work has shown that Futhark's objective performance is generally good [13], so we consider our results representative of the cost of adding bounds checking to already well-performing code.

4.1 Methodology

Each benchmark program is compiled and benchmarked with four different ways of handling bounds checks:

Without any checking our baseline.
Checked full bounds checking.
Unoptimised excludes the optimisations from Sect. 3.
Synchronous `clFinish()` after every kernel enqueuing.

For the latter three we report the relative slowdown compared to the baseline. Our experiments are run on two systems: an AMD Vega 64 GPU where we use Futhark's OpenCL backend, and an NVIDIA RTX 2080 Ti GPU where we use Futhark's CUDA backend. We use the `futhark bench` tool to perform the timing, and the OpenCL

[2] https://github.com/diku-dk/futhark-benchmarks.

Table 1 Benchmarks and datasets used for the measurements on Fig. 4

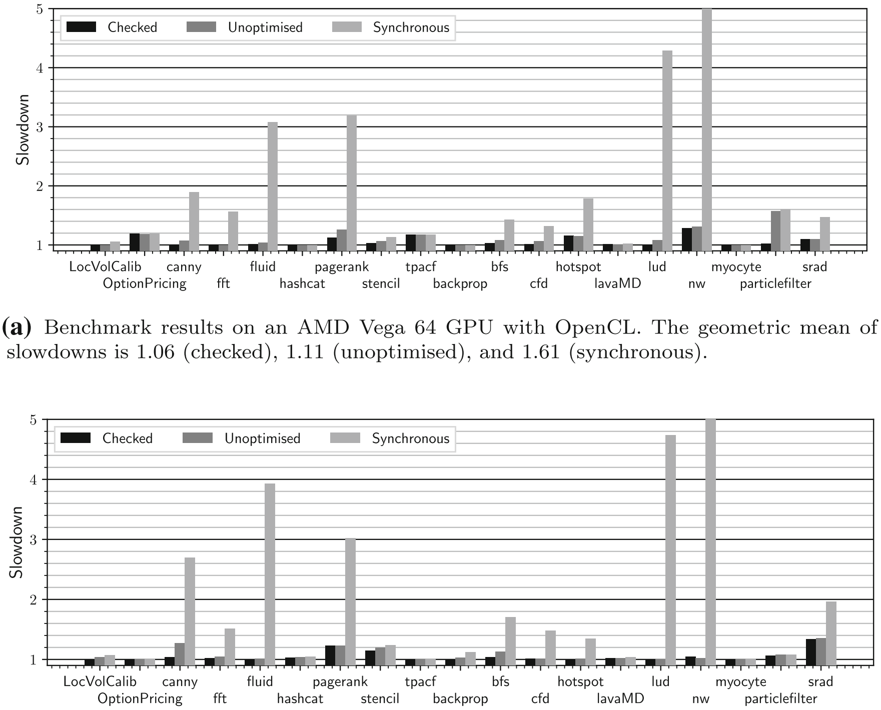
Benchmark	Dataset
FinPar	
LocVolCalib	Large
OptionPricing	Large
Accelerate	
Canny	512 × 512
fft	1024 × 1024
Fluid	Medium
Hashcat	Rockyou
Pagerank	Small
Parboil	
Stencil	Default
Tpacf	Large
Rodinia	
Backprop	Medium
bfs	graph1MW_6
cfd	fbcorr.domn.193K
Hotspot	1024
lavaMD	10 boxes
lud	2048
nw	large
Myocyte	Medium
Particlefilter	128 × 128 × 10 image, 400000 particles
srad	502 × 458 image

The names of datasets are from the original benchmark sources (hence the inconsistent naming), and have been chosen to be the largest available

backend for code generation. The timing does not include GPU driver setup and teardown, nor does it include copying the *initial* input data to the GPU, nor the *final* results from the GPU. All other CPU–GPU communication is counted. We report the average runtime of 10 runs for each benchmark.

4.2 Results

The results are shown on Fig. 4. The most obvious conclusion is that synchronous execution can have ruinous overhead; exceeding 5× for the *nw* benchmark, and exceeding 2× on five other benchmarks on the RTX2080. The most affected benchmarks are structured as a rapid sequence of kernels that each run for at most a few dozen microseconds. Halting the GPU after every kernel, rather than letting it process the



(a) Benchmark results on an AMD Vega 64 GPU with OpenCL. The geometric mean of slowdowns is 1.06 (checked), 1.11 (unoptimised), and 1.61 (synchronous).

(b) Benchmark results on an NVIDIA RTX2080 Ti GPU with CUDA. The geometric mean of slowdowns is 1.04 (checked), 1.07 (unoptimised), and 1.66 (synchronous).

Fig. 4 Runtime slowdown of performing bounds checking compared to not performing bounds checking. See Table 1 for the benchmark workloads. *Checked* is the full implementation with the optimisations listed in Sect. 3. *Unoptimised* is without the optimisations. *Synchronous* is with GPU synchronisation after every kernel enqueuing

queue on its own, adds a significant constant cost to every kernel, which slows down these benchmarks significantly. On the other hand, those benchmarks that run just a few large kernels, such as *OptionPricing*, are not significantly affected. The Vega 64 is slightly less hampered than the RTX 2080 Ti, which is likely because the Vega 64 is relatively slower, so the kernels run for longer on average.

The Sect. 3 optimisations have a relatively small impact on most benchmarks. The largest impact is on *particlefilter* on the Vega 64, where the optimisations reduce overhead from almost 1.6× to essentially nothing. The *bfs* benchmark is an excellent demonstration of bounds checking, as it implements a graph algorithm by representing the graph as several arrays containing indexes into each other. A compiler would have to be *sufficiently smart* to a very high degree to statically verify these index operations. At the same time, most of the GPU kernels are `map` or `scatter`-like operations with no communication between threads, so the Sect. 3.4 optimisation applies readily. The *srad* and *stencil* benchmarks suffer significantly under bounds checking on the RTX 2080 Ti. Both of these are stencil nine-point stencil computations, and are written in a way that prevents the Futhark compiler from statically resolving eight of the nine bounds checks that are needed for each output element.

5 Conclusions

We have demonstrated an implementation technique for implementing checking of array indexing and similar safety checks in GPU kernels generated from high-level array languages, even when the GPU programming API does not support abnormal termination.

Implementing the technique in a mature GPU-targeting compiler took only moderate effort, and our experiments show that the overhead of bounds checking has a geometric average of a relatively modest 6%, counting only those programs where checking is necessary in the first place. This is comparable to other work on bounds-checking C programs [1], although this comparison is admittedly not entirely fair, as a functional array language need not check indexing that arises from operations such as `map` and `reduce`. Nevertheless, our results suggest that bounds checking can be performed by default even in high-performance array languages.

Acknowledgements This research has been partially supported by a grant from the Independent Research Fund Denmark, under the research project *FUTHARK: Functional Technology for High-performance Architectures*.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th Conference on USENIX Security Symposium, USENIX Association, USA, SSYM'09, pp. 51–66 (2009)
2. Andreetta, C., Bégot, V., Berthold, J., Elsman, M., Henglein, F., Henriksen, T., Nordfang, M.B., Oancea, C.E.: Finpar: a parallel financial benchmark. ACM Trans. Archit. Code Optim. **13**(2), 18:1-18:27 (2016)
3. Bernecky, R., Scholz, SB.: Abstract expressionism for parallel performance. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, pp. 54–59 (2015)
4. Besard, T., Foket, C., De Sutter, B.: Effective extensible programming: unleashing Julia on GPUs. IEEE Trans. Parallel Distrib. Syst. **30**(4), 827–841 (2019)
5. Chakravarty, MM., Keller, G., Lee, S., McDonell, TL., Grover, V.: Accelerating haskell array codes with multicore GPUs. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, Association for Computing Machinery, New York, NY, USA, DAMP '11, pp. 3–14. https://doi.org/10.1145/1926354.1926358 (2011)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, JW., Lee, SH., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), IEEE, pp. 44–54 (2009)
7. Cunningham, D., Bordawekar, R., Saraswat, V.: GPU programming in a high level language: compiling x10 to cuda. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop, Association for Computing Machinery, New York, NY, USA, X10 '11. https://doi.org/10.1145/2212736.2212744 (2011)
8. Dijkstra, E.: Go to Statement Considered Harmful, pp. 27–33. Yourdon Press, USA (1979)
9. Erb, C., Greathouse, JL.: Clarmor: A dynamic buffer overflow detector for opencl kernels. In: Proceedings of the International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, IWOCL '18. https://doi.org/10.1145/3204919.3204934 (2018)
10. Fumero, JJ., Steuwer, M., Stadler, L., Dubach, C.: Just-in-time GPU compilation for interpreted languages with partial evaluation. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International

Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8–9, 2017, ACM, pp. 60–73. https://doi.org/10.1145/3050748.3050761 (2017)

11. Guo, J., Thiyagalingam, J., Scholz, SB.: Breaking the GPU programming barrier with the auto-parallelising sac compiler. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, Association for Computing Machinery, New York, NY, USA, DAMP '11, pp. 15–24. https://doi.org/10.1145/1926354.1926359 (2011)
12. Henriksen, T., Dybdal, M., Urms, H., Kiehn, AS., Gavin, D., Abelskov, H., Elsman, M., Oancea, C.: APL on GPUs: a tail from the past, scribbled in futhark. In: Proceedings of the 5th International Workshop on Functional High-Performance Computing, ACM, New York, NY, USA, FHPC 2016, pp. 38–43. https://doi.org/10.1145/2975991.2975997 (2016)
13. Henriksen, T., Serup, NGW., Elsman, M., Henglein, F., Oancea, CE.: Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI 2017, pp. 556–571. https://doi.org/10.1145/3062341.3062354 (2017)
14. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, PPoPP '19, pp. 53–67. https://doi.org/10.1145/3293883.3295707, (2019)
15. Hoare, C.A.R.: The emperor's old clothes. Commun. ACM **24**(2), 75–83 (1981). https://doi.org/10.1145/358549.358561
16. Holk, E., Newton, R., Siek, J., Lumsdaine, A.: Region-based memory management for GPU programming languages: enabling rich data structures on a spartan host. SIGPLAN Not. **49**(10), 141–155 (2014). https://doi.org/10.1145/2714064.2660244
17. Hsu, AW.: A data parallel compiler hosted on the GPU. PhD Thesis, Indiana University (2019)
18. Price, J., McIntosh-Smith, S.: Oclgrind: an extensible opencl device simulator. In: Proceedings of the 3rd International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, IWOCL '15. https://doi.org/10.1145/2791321.2791333 (2015)
19. Steuwer, M., Remmelg, T., Dubach, C.: Lift: A functional data-parallel IR for high-performance GPU code generation. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, IEEE Press, CGO '17, pp. 74–85 (2017)
20. Stratton, JA., Rodrigues, C., Sung, IJ., Obeid, N., Chang, LW., Anssari, N., Liu, GD., Hwu, WmW.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. In: Center for Reliable and High-Performance Computing, vol. 127 (2012)
21. Xi, H.: Dependent ml an approach to practical programming with dependent types. J. Funct. Program. **17**(2), 215–286 (2007). https://doi.org/10.1017/S0956796806006216