Technical Report

UCAM-CL-TR-997
ISSN 1476-2986

Number 997

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

CHERI-SIMT report: implementing capability memory protection in GPGPUs

Matthew Naylor, Alexandre Joannou,
A. Theodore Markettos, Paul Metzger,
Simon W. Moore, Timothy M. Jones

March 2025

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This work was supported by the UK EPSRC under the "CAPcelerate Project" (EP/V000381/1) and the "Chrompartments Project" (EP/X015963/1), both part of the Digital Security by Design (DSbD) Programme and the DSbDtech initiative.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-997*

Abstract

Governments are increasingly advising software manufacturers to employ memory-safe languages and technologies to combat adversarial attacks on modern computing infrastructure. This introduces pressures across the entire computing industry, including GPGPU vendors who provide implementations of unsafe C/C++-based languages, such as CUDA and OpenCL, for programming the devices they produce. One of the memory-safety technologies being recommended is *Capability Hardware Enhanced RISC Instructions* (CHERI). CHERI adds strong and efficient memory safety to underlying instruction-set architectures allowing continued, but memory-safe, use of C/C++-based languages on top. Another option being recommended is Rust, a memory-safe systems programming language that can viably replace C/C++ in some cases.

In this report, we evaluate the feasibility of incorporating CHERI into GPGPU architectures by extending a prototype, open-source, synthesisable, SIMT core and CUDA-like programming environment with support for CHERI. We present techniques to considerably ameliorate the costs of CHERI in SIMT designs, reducing register-file storage overheads from 103% to 7%, logic-area overheads by 44% to a cost comparable to one additional multiplier per vector lane, and execution-time overheads to 1.6%. By comparison, an experimental Rust port of the same GPGPU benchmark suite shows a 34% increase in execution time due to software bounds checking. With the proposed techniques, CHERI offers a viable path to strong and efficient GPGPU memory safety, while avoiding the need to replace established programming practices.

Contents

1 Introduction

Recent studies report that memory-safety bugs account for around 70% of security vulnerabilities in major software projects [12, 49, 71]. These bugs typically arise from mistakes involving C/C++ pointers, such as out-of-bounds accesses (violating *spatial* memory safety) and use-after-free errors (violating *temporal* memory safety). In response, government agencies around the world are increasingly advising software manufacturers to employ memory-safe languages and technologies, as set out in recent joint whitepapers [13, 14]. There are also calls to standardise principles and practices for software memory safety, allowing future systems to comply with agreed criteria for preventing this whole class of vulnerabilities [63, 77].

While the above-mentioned studies focus on CPU codebases, GPGPU codebases are also commonly written in C/C++-based languages, such as CUDA and OpenCL, and have been found to suffer from many of the same problems. For example, Erb et al. [23] report 13 cases of buffer overflows in a set of 175 GPGPU applications taken from standard benchmark suites. Separately, researchers have shown that buffer overflows on GPGPUs can lead to data corruption on the stack and heap, control-flow hijacking, code injection, and arbitrary code execution [21, 31, 48, 61]. A simple example of a buffer overflow in CUDA is shown in Figure 1.

One of the memory-safety technologies being mentioned both in the joint government whitepapers and the memory-safety standardisation call is *Capability Hardware Enhanced RISC Instructions* (CHERI) [75]. CHERI enhances existing instruction-set architectures by replacing integer memory addresses with *capabilities*. Capabilities augment integer addresses with *metadata* including bounds and permissions, enforcing constrained access to bounded memory regions. In addition to providing deterministic spatial memory safety using bounds, they also provide *referential integrity* (they cannot be forged, corrupted, or confused with non-capabilities), which in turn supports temporal memory safety [26, 27] as well as efficient software compartmentalisation [3, 78], limiting what attackers can do if they gain access through other kinds of vulnerabilities. A key attraction of CHERI is that pointers in historically unsafe C/C++ can be automatically compiled down to capabilities while requiring only minor changes (if any) to existing codebases.

Over the past decade, CHERI has been studied heavily in the context of CPUs [67, 76, 79] and various challenges have been overcome to reduce hardware and performance costs to satisfactory levels [37, 73, 80]. This has led to industrial implementations from Arm [5], Microsoft [3], and Codasip [15]. However, CHERI has not yet been applied to GPGPUs, which bring new challenges. Modern GPGPUs support hundreds of thousands of hardware threads leading to huge register files that account for a significant proportion of overall silicon area and power usage [28, 50]. CHERI increases both the pointer size and the architectural register size by a factor of two, potentially doubling the already-large register-file cost, which would be prohibitive. Indeed, prior work dismisses the use of CHERI on GPGPUs for precisely this reason [41]. Additionally, CHERI introduces a range of new instructions to manipulate capabilities and implementing these in every execution unit also incurs costs.

In this paper, we demonstrate techniques to considerably ameliorate the costs of CHERI in *Single Instruction, Multiple Thread* (SIMT) GPGPUs. We observe a large amount of redundancy in the metadata of capabilities between hardware threads, which can be readily exploited in SIMT architectures to reduce onchip storage overheads. We also observe that several instruc-

```
__global__ void overread() {
  int data = 0xda1a;
  int secret = 0xc0de;
  int *ptr = &data;
  printf("%x\n", ptr[1]);
}
```

Figure 1: Simple example of a buffer overread in a CUDA GPGPU kernel. In practice, this function prints the value of `secret`. Although `ptr` points to `data`, it is accessed out of bounds to obtain the value of a different variable.

tions introduced by CHERI are rarely found in the hot code paths of GPGPU kernels, allowing them to be implemented in a shared-function unit without impacting run-time performance. Exploiting these observations, we develop an efficient implementation of CHERI on top of an existing, prototype, open-source, synthesisable, SIMT GPGPU (SIMTIGHT) and CUDA-like C++ programming environment (NOCL) [53]. Our main results are as follows:

- We reduce the register-file storage overheads of CHERI from 103% to 14% by exploiting value regularity in capability metadata. With basic compiler support to limit the number of registers used to hold capabilities, we forecast that this overhead could be reduced to 7% without impacting run-time performance. Estimating that the register file accounts for less than half of the total onchip storage in a full GPGPU design, the overall storage overhead of CHERI would fall below 3.5%.
- We reduce the logic-area overheads of CHERI by 44% per streaming multiprocessor (SM) by moving non-critical logic into a shared-function unit. The absolute overhead is comparable to (but slightly larger than) the cost of an additional multiplier per vector lane.
- We find the run-time overhead of CHERI to be 1.6% on average across a range of GPGPU benchmarks — lower than reported CHERI overheads for CPU workloads.
- All GPGPU benchmarks are simply recomplied to achieve full spatial memory safety and referential integrity. No source code changes to the benchmarks are required.

With these optimisations, CHERI offers a viable path to strong and efficient GPGPU memory safety for established C/C++-based languages such as CUDA and OpenCL. For comparison, we develop and evaluate an experimental Rust port of NOCL for SIMTIGHT, which shows an average 34% overhead for software bounds checking on the same GPGPU benchmarks. We also compare qualitatively against existing approaches to GPGPU memory safety, covering both hardware support and the use of memory-safe languages.

2 Background

2.1 Single Instruction, Multiple Threads (SIMT)

SIMT is a parallel execution model, popularised by NVIDIA and AMD GPGPUs, that combines the flexibility of a multi-thread programming model with the efficiency of SIMD hardware. The idea is to execute multiple hardware threads in lockstep with the aim of exploiting regularity between them. Known collectively as a *warp* (or *wavefront*), these threads can exhibit three main kinds of regularity [39]:

- *Control-flow regularity*, where threads in a warp follow the same path through the program. This allows the costs of fetching and decoding instructions to be amortised over multiple execution units.

- *Memory-access regularity*, where threads in a warp access neighbouring addresses in memory. This allows a large number of narrow memory requests to be coalesced into a small number of wide requests, which can be more efficiently handled by the memory subsystem.

- *Value regularity*, where threads in a warp compute the same or similar intermediate values. This allows common data and computations to be shared between threads, reducing onchip storage costs and energy consumption.

While SIMT processors rely on inter-thread regularity to achieve optimal performance, they nevertheless permit general, independent, scalar computation in each thread. When threads in a warp *diverge* and take different paths through the program they can no longer execute in lockstep, leading to lower utilisation of the SIMD execution units. However, SIMT aims to *reconverge* these threads at the earliest opportunity to restore regularity and performance.

In addition to executing multiple threads per warp in parallel, SIMT processors also execute multiple warps concurrently, frequently context-switching between them (every cycle or every few cycles) to mask memory and compute latencies. A SIMT processor that executes multiple warps in this manner is typically referred to as *streaming multiprocessor*, or SM for short. NVIDIA's latest GA100 GPGPU supports 128 SMs, with up to 64 warps per SM and 32 threads per warp, yielding over 250K hardware threads in total. Each of these hardware threads has access to its own set of private registers. The GPGPU register file can therefore require many megabytes of fast onchip storage, accounting for a major proportion of overall silicon area and power usage [28, 50].

It can be convenient to reuse SIMD terminology when working with SIMT. Although SIMT involves execution of scalar threads, it is possible to view each scalar value in a thread as an element of a *vector* value in a warp. Similarly, it is possible to view the scalar execution unit for each thread in a warp as a *vector lane*.

In modern GPGPUs, the number of execution units (or vector lanes) per SM typically equals the number of threads per warp, but this is not strictly necessary. Early NVIDIA GPGPUs shared 8 execution units across 32 threads in a warp, serialising each instruction over 4 consecutive cycles.

2.2 GPGPU Programming

CUDA and OpenCL are the most widely used languages for programming GPGPUs today. These languages are both based on C/C++ and are broadly similar. In this report we focus on CUDA, which is more actively maintained. The main idea of CUDA is to let the programmer define a function, known as a *kernel*, that will be simultaneously invoked by multiple

```
__global__ void histogram(int len, unsigned char* in, int* out) {
  // Allocate bins in shared local memory (shared by threads in a block)
  __shared__ int bins[256];

  // Initialise bins
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
      bins[i] = 0;

  // Synchronisation barrier for threads in a block
  __syncthreads();

  // Accumulate bins
  for (int i = threadIdx.x; i < len; i += blockDim.x)
      atomicAdd(&bins[in[i]], 1);

  __syncthreads();

  // Write bins to global memory
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
      out[i] = bins[i];
}
```

Figure 2: CUDA kernel to compute a 256-bin histogram. Given an `len`-element array `in` of bytes, the kernel computes a 256-element array `out` such that `out[`b`]` contains the number of occurrences of the byte b in the array `in`. This kernel uses a single thread block and is therefore capable of running on at most one SM. It can be generalised to multiple thread blocks by arranging each thread block to compute a histogram for a different portion of the input array, and then merging these histograms together.

CUDA threads in parallel. Each CUDA thread determines what it should do based on its *thread id* within a 3D *block* (`threadId.x`, `threadId.y`, `threadId.z`) and its *block id* within a 3D *grid* (`blockId.x`, `blockId.y`, `blockId.z`). Thread blocks (of size `blockDim.x` × `blockDim.y` × `blockDim.z`) permit fast barrier synchronisation and efficient communication via shared local memory, but are limited to at most 1,024 or 2,048 threads. Meanwhile, grids can be of any size (`gridDim.x` × `gridDim.y` × `gridDim.z`) but are limited to less-efficient mechanisms for barrier synchronisation and communication. As an example, Figure 2 shows a simple CUDA kernel for computing 256-bin histograms. CUDA threads are mapped onto hardware threads at run time, and kernels are executed on as many SMs as possible to maximise use of GPGPU resources.

2.3 Value Regularity

Collange et al. first identified the prevalence of value regularity in SIMT workloads [19]. They use the term *uniform vector* to refer to a variable that has the same value in every thread in a warp, and *affine vector* to refer to a variable whose value is of the form $base + t \times stride$ for each thread t in a warp with a fixed *base* and *stride*. A uniform vector is a special case of an affine vector with a zero stride. The authors report that, on average over a range of CUDA benchmarks running in simulation, 27% of vectors read from the register file, and 15% of vectors written, are uniform. These numbers rise to 44% and 28% respectively for affine vectors.

The prevalence of value regularity in SIMT workloads can by understood by looking at the

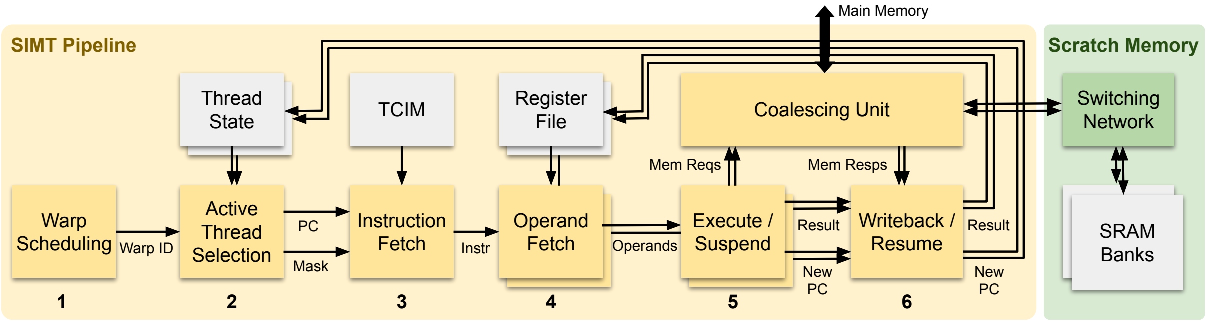


Figure 3: Diagram of the SIMTIGHT streaming multiprocessor (SM) [53], including the pipeline, tightly-coupled instruction memory (TCIM), coalescing unit, and scratchpad memory. Double boxes represent components containing logic or storage that is replicated per vector lane, and double lines represent per-lane wiring.

data-parallel programming frameworks in which they are implemented, such as CUDA and OpenCL. In CUDA, each thread typically decides which part of the input to process based on its thread index within a block, and its block index within a grid. Threads in the same warp always reside in the same block, hence calculations on the block index will involve uniform vectors. Similarly, threads in the same warp will have consecutive thread indices, hence calculations on the thread index will involve affine vectors with a unit stride. To compound this, the CUDA programming guide [60] recommends that memory addresses for loads and stores should be the same (uniform) or consecutive (affine) across a warp, where possible, to maximise performance.

Value regularity can be exploited to reduce onchip storage requirements by storing uniform and affine vectors in a compact form. It can also be exploited to reduce energy requirements, or improve instruction throughput, by processing uniform and affine vectors in a specialised affine data path (with a single execution unit), separate from the general vector data path (with multiple execution units). These optimisations can be achieved using a technique known as *scalarisation*, which either operates *statically* in the compiler with instruction-set support [1, 9, 17, 36, 43, 72, 82] or *dynamically* in the microarchitecture [19, 29, 39, 42, 46, 53, 64, 81].

2.4 The SIMTIGHT GPGPU

Over the past decade, open-source GPGPU hardware has emerged as a promising platform for research and development. While early designs were based either on proprietary instruction sets [6], preventing free deployment of hardware, or custom instruction sets [2, 4, 8], with no access to a mature software stack or compiler, recent designs have started to employ the open RISC-V standard [16, 53, 65, 70], avoiding both problems. Researchers have shown that RISC-V is well-suited to SIMT execution, and how to extend RISC-V to support graphics. They have also demonstrated OpenGL, OpenCL, and CUDA-like software stacks on top of SIMT-style RISC-V GPGPUs. SIMTIGHT [53] is the latest in a line of such GPGPUs, which we use as a basis for the work presented in this report.

SIMTIGHT implements RISC-V's `rv32ima_zfinx` profile, i.e., a 32-bit machine with integer, multiply and divide, atomics, and single-precision floating-point support, with a merged integer and floating-point register file. The streaming multiprocessor (SM) component of SIMTIGHT is depicted in Figure 3 and has the following features:

- It is parameterised by the number of warps and the number of threads per warp, the latter of which is equivalent to the number of vector lanes as all threads in a warp can execute an instruction in parallel.

- It employs a 6-stage processor pipeline fed by a barrel scheduler that switches between warps on every cycle. At most one instruction per warp is present in the pipeline at any time, avoiding data and control hazards.

- Each thread in a warp has its own program counter (PC), supporting control-flow divergence. The Active Thread Selection stage of the pipeline determines a subset of these threads which have the PC. The instruction at this common PC is fetched and decoded once, and then executed by all active threads in the warp, exploiting control-flow regularity. Thread convergence is achieved by prioritising the selection of threads residing at the deepest nesting level in the structured control-flow graph [18]. Further details about convergence can be found in the SIMTIGHT paper [53] and the NOCL manual [52].

- Warps executing a multi-cycle instruction are suspended in the execute stage and resumed in the writeback stage without blocking the pipeline, tolerating high-latency operations such as memory loads and floating-point operations.

- Memory-access regularity is exploited using a coalescing unit that tries to pack memory requests from each vector lane into a smaller set of wider main-memory accesses using coalescing rules similar to those found in early NVIDIA Tesla devices [45].

- Value regularity is exploited using a set of microarchitectural techniques referred to as *advanced dynamic scalarisation* [53]. This includes *register-file compression* to reduce register-file storage requirements, a key feature of SIMTIGHT that we exploit in our implementation of CHERI, which is is introduced in detail in Section 3.1.

- Efficient communication between hardware threads is facilitated through a scratchpad memory supporting parallel random access. This is implemented as a set of SRAM banks and a fast switching network. The scratchpad is critical for efficient implementation of `__shared__` memory in CUDA and `__local` memory in OpenCL.

The main limitation of SIMTIGHT is that it currently supports only a single SM. This is sufficient for the work described in this report, where the majority of changes relate to components within an SM rather than the memory subsystem that connects SMs together. However, this does mean that the overheads we report are relative to a single SM rather than a full GPGPU design with multiple SMs and an advanced shared-memory subsystem.

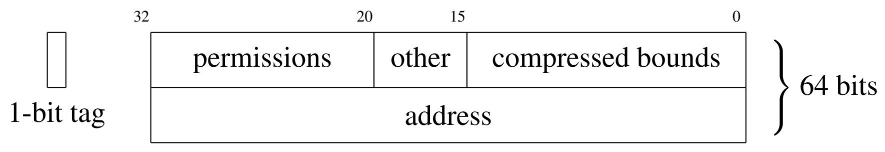
The SIMTIGHT distribution ships with a programming API called NOCL [52] that supports writing CUDA-like compute kernels in plain C++ (no special compute language is required). It also includes a suite of benchmark compute kernels written in NOCL. A sample NoCL kernel to compute 256-bin histograms is shown in Figure 4. The authors report high IPC in many benchmarks (Figure 5) as well as high performance density on FPGA compared to other open-source GPGPUs.

```
struct Histogram : Kernel {
  // Parameters
  int len; unsigned char* in; int* out;

  // Histogram bins in shared local memory
  int* bins;

  void init() {
    declareShared(&bins, 256);
  }

  void kernel() {
    // Initialise bins
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
      bins[i] = 0;

    __syncthreads();

    // Update bins
    for (int i = threadIdx.x; i < len; i += blockDim.x)
      atomicAdd(&bins[in[i]], 1);

    __syncthreads();

    // Write bins to global memory
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
      out[i] = bins[i];
  }
};
```

Figure 4: NoCL kernel written in plain C++ to compute the 256-bin histogram of a given byte array using a single thread block [53]. It is almost identical to the CUDA version of the same kernel, shown in Figure 2. Full source code for this benchmark, including host-side code, can be found in Appendix A.

2.5 Capability Hardware Enhanced RISC Instructions (CHERI)

CHERI extends conventional instruction-set architectures (MIPS, RISC-V, ARM, and x86) with *capabilities*. In the 32-bit RISC-V architecture, which we focus on in this report, CHERI replaces 32-bit machine-word addresses with 64+1-bit capabilities. A capability captures an address together with bounds and permissions, and can be stored in registers or in memory. The bit representation of a 64+1-bit capability is as follows.



When using a capability to access memory (e.g., via load and store instructions), CHERI requires the address to lie within the bounds, throwing an exception if that is not the case. This is the basis for enforcing spatial memory safety. CHERI then provides instructions to manipulate capabilities, such as modifying the address (pointer arithmetic), reducing the permissions, and narrowing the bounds. When manipulating capabilities, the address is allowed to wander out

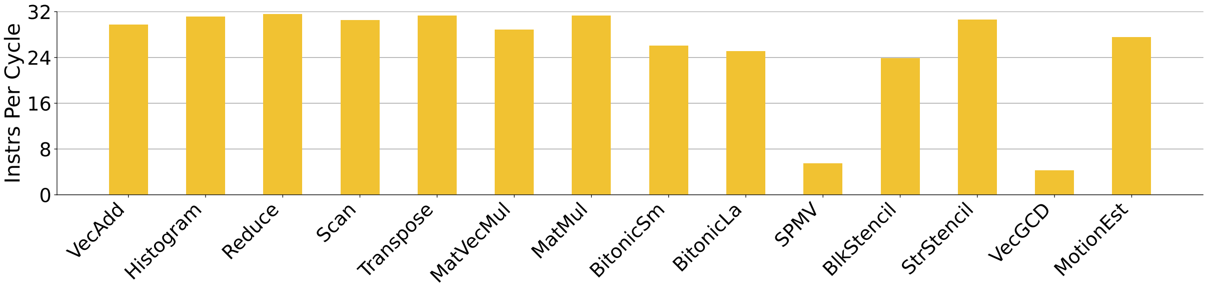


Figure 5: Instruction throughput for a single 32-lane SIMTight SM [53]. Workloads achieve an IPC approaching the number of vector lanes, where expected.

of bounds to some extent. This is necessary to implement C/C++ pointers, which are allowed to point one byte beyond the end of an object and, in practice, often point further away than that [11].

A key property of CHERI is that capabilities are *unforgeable*: the only way to create a capability is to derive one from an existing capability using CHERI instructions, and doing so can never increase the bounds or permissions of the original. Consequently, the only memory that can be accessed by software running on a CHERI processor is that which is transitively reachable from capabilities stored in the register file. To achieve non-forgeability, CHERI stores a hidden *tag bit* for every register, and for every 64-bit word in memory, to distinguish valid capabilities from normal data. This makes it impossible to write arbitrary data to a register, or to memory, and subsequently interpret that data as a valid capability. It also makes pointers precisely distinguishable from non-pointers, permitting garbage collection and revocation [26, 27].

The tag bits of capabilities in memory are typically implemented by storing them in a reserved region of memory that is not architecturally addressable. A component called the *tag controller*, placed in front of main memory, ensures that each addressable 64-bit word and its corresponding tag bit are accessed atomically by processors. The tag controller includes a *tag cache* to optimise access to tag bits. It turns out that the miss rate of the tag cache, and hence the overhead of accessing tag bits, can be reduced to almost zero in practice by exploiting the fact that blocks of memory (cache lines or pages) will often not hold any capabilities at all, allowing a highly compact representation in the tag cache [37].

To keep the size of a capability to just 64 bits, CHERI represents the bounds in a compressed format known as *CHERI Concentrate* [80]. Specifically, a 32-bit lower bound and a 33-bit upper bound are together stored in just 15 bits by encoding them in a floating-point-like format relative to the address. The hardware costs of bounds compression are discussed in Section 3.3.

The properties provided by CHERI are *deterministic* and *enforced*, not subject to probabilities or bypassing.

3 Design and Implementation

In this section, we present the main changes needed to efficiently implement CHERI in the SIMTight GPGPU. Specifically, we implement a large subset of version 9 of the 32-bit CHERI instruction set [76], as shown in Figure 6.

Get/clear tag bit	
CGetTag	rd, cs1
CClearTag	cd, cs1

Get/reduce permissions	
CGetPerm	rd, cs1
CAndPerm	cd, cs1, rs2

Get/set bounds	
CGetBase	rd, cs1
CGetLen	rd, cs1
CSetBounds	cd, cs1, rs2
CSetBoundsImm	cd, cs1, imm
CSetBoundsExact	cd, cs1, rs2

Load/store via capabilities	
CL[BHW][U]	rd, cs1, imm
CS[BHW]	rs2, cs1, imm

Load/store capabilities	
CLC	cd, cs1, imm
CSC	cs2, cs1, imm

Get/set/increment address	
CGetAddr	rd, cs1
CSetAddr	cd, cs1, rs2
CIncOffset	cd, cs1, rs2
CIncOffsetImm	cd, cs1, imm

Other	
AUIPCC	cd, imm
CJALR	cd, cs1, imm
CJAL	cd, imm
CMove	cd, cs1
CGetType	rd, cs1
CGetSealed	rd, cs1
CGetFlags	rd, cs1
CSetFlags	cd, cs1, rs2
CSealEntry	cd, cs1
CSpecialRW	cd, cs1, imm
CRRL	rd, rs1
CRAM	rd, rs1

Figure 6: List of CHERI instructions implemented in SIMTIGHT (excluding atomics). CHERI extends each 32-bit general-purpose register with 33-bits of metadata. Operands *rd*, *rs1*, and *rs2* refer to the 32-bit general-purpose portion of a register while operands *cd*, *cs1*, and *cs2* refer to the full 65-bit contents. When an instruction writes to *rd*, the capability metadata for that register is set to a *null value* with the tag bit cleared.

3.1 Register File

Adapting a 32-bit RISC-V implementation to support CHERI requires extending every 32-bit general-purpose register to 65 bits. This is potentially a large cost in a streaming multiprocessor with thousands of hardware threads, each with their own set of private registers. However, our hypothesis is that there is likely to be a lot of value regularity in the capability metadata between threads executing in lockstep: threads accessing different elements of the same array at the same time will likely involve the same bounds and permissions. Before trying to exploit this hypothesis, it is first useful to look at SIMTIGHT's existing register-file compression mechanism in more detail.

A single SIMTight streaming multiprocessor contains $32 \times \mathit{numWarps} \times \mathit{numThreadsPerWarp}$ architectural registers or, equivalently, $32 \times \mathit{numWarps}$ architectural *vector* registers (each scalar register in a thread can be viewed as an element of vector register in a warp). SIMTIGHT's compressed register file aims to reduce onchip storage requirements by exploiting the property that vector registers will often hold uniform or affine vectors that can be stored compactly. It detects these uniform and affine vectors at run time using an array of comparators in the register-file write path (cheaper inference-based mechanisms for detecting uniform and affine vectors are also possible [19, 39, 81] but are not yet supported in SIMTIGHT). Uniform and affine vectors are then stored in a scalar register file (SRF) while general (non-compressible) vectors are allocated on-demand in a larger, size-constrained vector register file (VRF). For every architectural vector register, the SRF either holds a compressed vector (a $\mathit{base} + \mathit{stride}$ pair) or a pointer to a register in the VRF. The size of the VRF can be set arbitrarily. Currently in SIMTIGHT it is chosen statically at synthesis time based on experimental evaluation, but in principle it can be set dynamically, which is useful if, as in modern NVIDIA GPGPUs, the physical memory implementing the register file is shared with other storage structures such as the scratchpad and L1 cache [28]. VRF overflow is handled in hardware by dynamically spilling vector registers to main memory. The overall structure of the compressed register file is shown and explained in Figure 7.

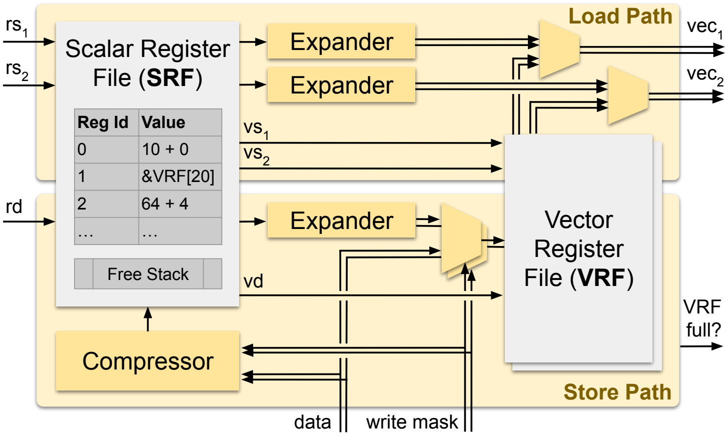


Figure 7: Overview of SIMTight's compressed register file. Registers rs_1 and rs_2 are looked up to produce vectors vec_1 and vec_2 respectively. The active elements of vector *data* (as specified by *write mask*) are written to register *rd*. If any of the registers rs_1, rs_2, and *rd* are not held in the SRF then the SRF emits the locations vs_1, vs_2, and *vd* of these registers respectively in the VRF. The *Compressor* attempts to transform a vector to a *base* + *stride* pair that can be stored in the SRF, while the *Expander* performs the inverse transformation. For vectors that cannot be compressed, the *Free Stack* tracks unused locations in the VRF where they can be stored. If this stack becomes near-empty, the *VRF full* flag is asserted, triggering the pipeline to spill registers from the VRF to main memory.

The SIMTight authors report that a compressed register file with a quarter-sized VRF (i.e., a VRF big enough to hold a quarter of all architectural vector registers) has a minimal impact on run-time performance while reducing register-file storage requirements by 68% per 2,048-thread SM.

3.2 Metadata Register File

To support CHERI in the register file, we instantiate two compressed register files: a 32-bit general-purpose register file and a new 33-bit capability-metadata register file. This has the advantage that integer addresses and capability metadata are compressed separately: if we have a vector of capabilities that all have the same metadata but different (non-affine) addresses, then the metadata can still be compressed even though the addresses cannot. We enable the detection of only uniform (not affine) vectors in the metadata register file as the notion of a stride does not really exist for capability metadata.

The main drawback of this simple approach is *fragmentation*: there are two VRFs, each capable of holding on only one kind of vector (data or metadata), and if one becomes full then spilling will occur even if there is space available in the other. Furthermore, the SIMTight baseline enforces a minimum VRF capacity of four vector registers per thread and capability metadata may be more compressible than that. We therefore implement a *shared VRF* between the general-purpose and capability-metadata register files. To avoid additional read ports on the shared VRF, we serialise data and metadata accesses, i.e., accessing a register that requires both an uncompressed data vector *and* and uncompressed metadata vector will result in a pipeline stall. Our hypothesis is that this will not happen very often due to inter-thread regularity.

As shown in Figure 7, the baseline SRF requires three read ports, two in the load path and one

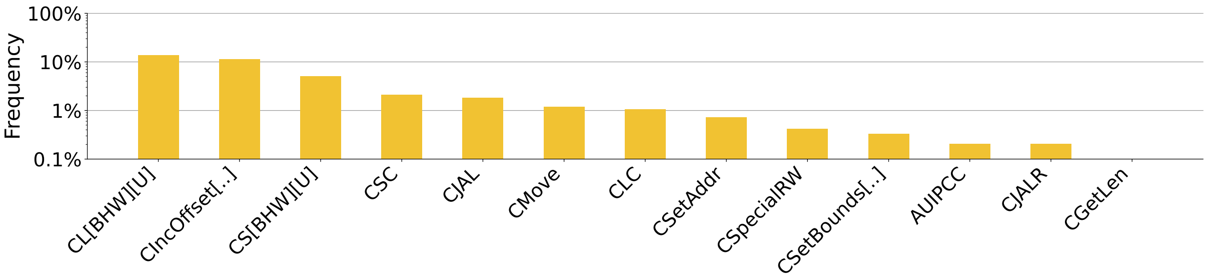


Figure 8: Average execution frequency of CHERI instructions on GPGPU workloads relative to total instructions executed, obtained using the experimental setup introduced in Section 4. CHERI instructions that are not shown here are not executed at all in the analysed workloads.

in the store path. SIMTIGHT implements this using two instances of a two-read-port SRAM primitive, with each instance holding identical data. This can be avoided in the capability-metadata SRF. As shown in Figure 6, the vast majority of CHERI instructions use only one capability source operand (i.e., *cs1* and not *cs2*). Only the CSC instruction to store a capability via a capability refers to both capability source operands. We therefore reduce the number of read ports on the capability-metadata SRF in exchange for taking an extra cycle to implement CSC. This means that the capability-metadata SRF essentially uses half the amount of storage as the baseline SRF. As shown in Figure 8, the execution frequency of the CSC instruction is quite low, around 2%.

The baseline register file restricts itself to *total* scalarisation: for a vector to be compressible, *all* elements must satisfy the uniform or affine requirements. One can envisage a generalisation of this where, for example, a vector can be partitioned into two different uniform or affine vectors and still held more compactly than an uncompressed vector. This would be useful in the presence of control-flow divergence, when a uniform or affine vector gets partially overwritten with a different uniform or affine vector. To implement this, each SRF entry would need to store an additional *base* + *stride* pair and a bit mask denoting which partition each vector element belongs to. Unfortunately, we have found that this SRF cost generally outweighs the associated savings in the VRF. However, the situation is slightly different for the capability-metadata SRF. First, it is half the size of the baseline SRF so increasing its size by a constant factor is not as expensive in absolute terms. Second, when a register holds an integer or floating-point value, rather than a capability, which is often the case, the metadata for that register is known to be a constant *null* value. This raises the possibility of extending the SRF with just a mask denoting which vector elements are null, supporting a form of partial scalarisation. We have implemented this as an optional feature in the metadata register file, which we refer to as the *null-value optimisation* (NVO). When a uniform vector is partially overwritten with a null vector, or vice-versa, that vector remains in the SRF. Furthermore, when a partially uniform vector is partially overwritten with a null vector or the same partially uniform vector, that vector also remains in the SRF.

3.3 Pipeline

Besides increasing the register size, another cost of adding CHERI to SIMTIGHT lies in the additional logic required to implement CHERI instructions in the pipeline and, in particular,

```
– In-memory capability format including tag bit
type  CapMem       =  Bit  65
– In-pipeline (partially decompressed) capability format
type  CapPipe      =  Bit  91
– Memory access width: 2^0, 2^1, 2^2, or 2^3 bytes
type  AccessWidth  =  Bit  2

Frequently needed
– Convert from the in-memory format
fromMem          ::  CapMem → CapPipe                              (46 ALMs)
– Convert to the in-memory format
toMem            ::  CapPipe → CapMem                               (0 ALMs)
– Set the address, invalidating the capability if the address is too far out-of-bounds
setAddr          ::  (CapPipe, Bit  32) → CapPipe                 (106 ALMs)
– Check that an access of given width from given capability is within bounds
isAccessInBounds ::  (CapPipe, AccessWidth) → Bit 1                (25 ALMs)

Infrequently needed
– Return the base (lower bound) of the capability
getBase          ::  CapPipe → Bit 32                              (50 ALMs)
– Return the length of the capability
getLength        ::  CapPipe → Bit 33                              (20 ALMs)
– Return the top (upper bound) of the capability
getTop           ::  CapPipe → Bit 33                              (78 ALMs)
– Return a capability of given length whose base is the address of given capability
setBounds        ::  (CapPipe, Bit  32) → CapPipe                 (287 ALMs)
```

Figure 9: Key functions of the CHERICAPLIB library [66] to handle compressed bounds in 64+1-bit capabilities. The logic area requirement of each function in ALMs (Intel Stratix 10 Adaptive Logic Modules) is listed on the right-hand side. As a point of reference, a 32-bit multiplier requires 567 ALMs. As shown, the *isAccessInBounds* function can check against partially decompressed bounds much more cheaply than fully decompressing the bounds via *getBase* and *getTop* and then using two address-width comparators.

to handle compressed bounds in capabilities. We use a standard library implementation of the CHERI Concentrate compressed bounds format [80] called CHERICAPLIB [66] whose main functions, along with their logic-area costs, are summarised in Figure 9. While CHERICAPLIB functions for getting and setting bounds are quite expensive, the CHERI instruction histogram in Figure 8 shows that instructions to get and set bounds are not frequently executed in GPGPU workloads. This motivates the separation of CHERICAPLIB functions into those that are frequently needed (such as pointer arithmetic and bounds checking) and those that are infrequently needed (such as getting and setting bounds). The former can be instantiated per vector lane (fast path) while the latter can be instantiated per SM in a shared-function unit (slow path).

The shared-function unit (SFU) is a common feature in GPGPU designs and indeed SIMTIGHT includes one that is used to implement floating-point square root and division. SIMTIGHT's SFU connects to every vector lane via a request serialiser and a response deserialiser. To support CHERI instructions in the SFU, we increase the size of SFU requests and responses to hold capability-sized operands and results respectively. This, in turn, increases the logic needed for serialisation and deserialisation but, overall, area is substantially reduced by moving logic out

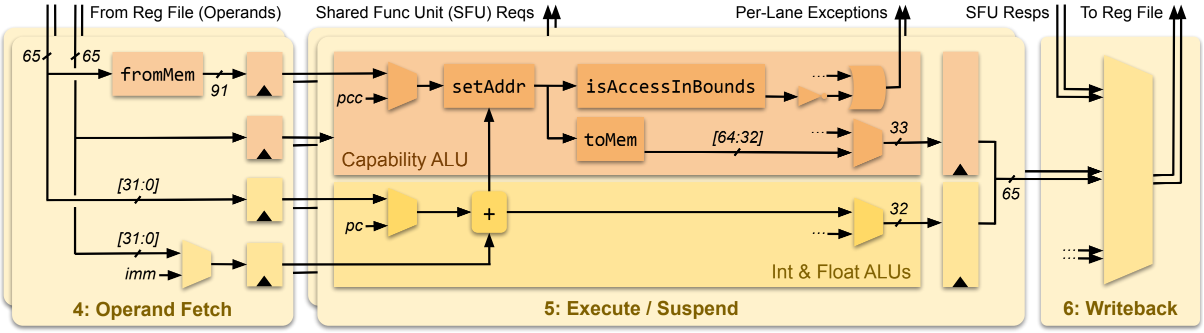


Figure 10: SIMTIGHT pipeline modifications to support CHERI. The output of the integer ALU's adder either contains the address result for CIncOffset[..]/CSetAddr, the address to access for CL[..]/CS[..], or the address to jump to for CJAL[..]. This address is fed into *setAddr* to yield a capability that is either written to a result register via *toMem*, fed to *isAccessInBounds* for bounds checking, or written to the PCC.

of the per-lane ALUs. We implement the CGetBase, CGetLen, CSetBounds[..], CRRL, and CRAM instructions in the SFU. All other CHERI instructions are implemented per vector lane using four CHERICAPLIB function calls, as shown in Figure 10.

In CHERI, the program counter is also considered to be a capability. Accordingly, we extend SIMTight's per-thread program counters (PCs) to be program-counter capabilities (PCCs). In the Active Thread Selection stage of the pipeline, we require that the chosen threads not only have the same PC but the same PCC. This means that only a single program-counter bounds check is required per SM. Nevertheless, these changes do increase logic area and, in software, we do not currently exploit the ability to change program-counter metadata dynamically in GPGPU kernels. We therefore provide an optional feature whereby PC metadata is set once per warp at kernel-invocation time but never changed. This allows Active Thread Selection to disregard PC metadata. We refer to this feature as the *static PC metadata restriction*.

3.4 Memory Subsystem

SIMTIGHT's memory subsystem supports 8-bit, 16-bit, and 32-bit accesses natively. To implement 64-bit (capability width) accesses, we use *multi-flit transactions* whereby a series of contiguous memory requests terminated by an *is-final* bit are treated atomically by the memory subsystem. A 64-bit access is then achieved using two inseparable 32-bit accesses. This avoids increasing the data-path width and associated logic in the memory subsystem in exchange for a two-cycle capability access time. As shown in Figure 8, the CLC and CSC instructions for loading and storing capabilities are executed fairly infrequently. The logic required to serialise and deserialise 64-bit requests and responses respectively is placed between the pipeline and the coalescing unit.

CHERI requires a 1-bit tag to be maintained for every naturally aligned 64-bit value in memory, indicating whether or not that 64-bit value holds a capability or not. As SIMTight's memory subsystem is natively 32-bit, we opt to maintain a 1-bit tag for every 32-bit naturally aligned value for simplicity (not necessity). We maintain the invariant that for a 64-bit capability to be valid, the tag bits of both its upper and lower halves must be set. To allow capabilities to be stored in scratch memory, we extend the data width of each onchip SRAM scratchpad

Benchmark	Description	Source
VecAdd	Vector addition	[58]
Histogram	256-bin histogram calculation	[57]
Reduce	Vector summation	[57]
Scan	Parallel prefix sum	[54]
Transpose	Matrix transpose	[57]
MatVecMul	Matrix × vector multiplication	[58]
MatMul	Matrix × matrix multiplication	[57]
BitonicSm	Bitonic sorter (small arrays)	[58]
BitonicLa	Bitonic sorter (large arrays)	[58]
SPMV	Sparse matrix × vector multiplication	[7]
BlkStencil	Block-based stencil computation	In house
StrStencil	Stripe-based stencil computation	In house
VecGCD	Vectorised greatest common divisor	In house
MotionEst	Motion estimation	In house

Figure 11: NoCL benchmark suite.

bank from 32 to 33 bits. To allow capabilities to be stored in main memory, we follow the approach taken by existing CHERI-enabled CPUs: tag bits are stored in a reserved region and a tag controller is placed just in front of main memory providing the illusion of atomic access to each value and its tag bit [37, 67, 79].

Compressed uniform and affine vectors can be passed from the register file to the memory subsystem for further storage savings. As a proof-of-concept, SIMTight includes a *compressed stack cache* in the coalescing unit to optimise compiler-inserted register spills of uniform and affine vectors. This reduces memory bandwidth utilisation while requiring only a small amount of additional onchip storage. We extend the compressed stack cache to be able to store capabilities. As we compress the metadata separately from the integer address, it is natural to allow the metadata to be cached even if the address cannot. This breaks the atomicity of capability writes, but we allow it on the basis that stack data is private and not shared between threads (a property that can be enforced using CHERI itself), making it impossible to observe a partially written capability.

4 Evaluation

In this section, we evaluate the overheads of adding CHERI to SIMTight and the impact of our proposed optimisations. All artefacts used for evaluation are available online [51]. Instructions for reproducing the main results are available in Appendix B.

4.1 Experimental Setup

We use the suite of 14 CUDA-like NoCL benchmark programs shipped as part of the standard SIMTight distribution [51] and listed in Figure 11. To compile the benchmarks, we use the CHERI fork of Clang 13 (the latest compiler supporting CHERI-RISC-V at the time), both when targeting RISC-V and CHERI-RISC-V. For a fair comparison, we disable *scalar evolution* (SCEV) of pointers, a compiler optimisation that is not yet supported when target-

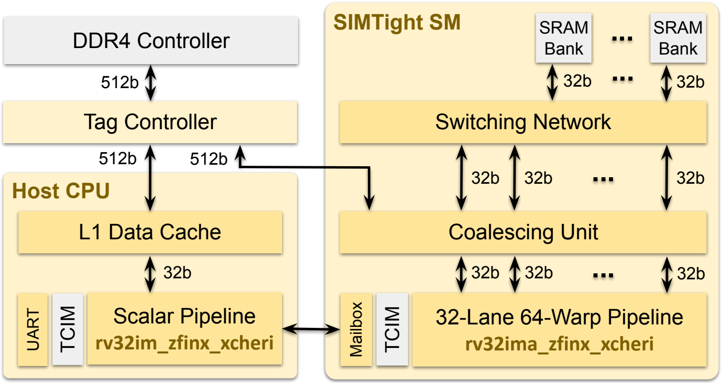


Figure 12: Diagram of the SIMTIGHT evaluation SoC, with data-bus widths (excluding tag bits).

ing CHERI-RISC-V but which is expected to be available in future. To support CHERI, some minor changes to the NOCL library are needed, such as setting the bounds of the stack and dynamically allocated buffers, but the benchmarks themselves do not require any source-code modifications at all.

We use the standard optimisation level `-O2` but force the compiler to inline the `kernel()` method of every NOCL compute kernel. This avoids function-call overhead in the NOCL inner loop, which invokes this method for every thread in a block, and every block in a grid. Aggressive inlining is standard when compiling GPGPU code, e.g., all device functions in CUDA are inlined by default [60]. In Section 4.4, we discuss some of the advantages of inlining in more detail.

We obtain all of our results on a Terasic DE10-Pro development board with an Intel Stratix-10 FPGA holding a single SIMTIGHT SM connected to a DDR4 DIMM, a tightly-coupled instruction memory (TCIM), and a CHERI-enabled host CPU, as depicted in Figure 12. Following modern NVIDIA devices, and prior work on SIMTIGHT, we use 64 warps and 32 threads per warp providing 2,048 hardware threads in total per SM. This number of warps is sufficient to mask the latency of DDR4 memory on FPGA, achieving good performance without caches. While the absence of caches reduces the accuracy of our setup for predicting ASIC performance, we show in Section 4.4 that our reported CHERI overheads are not affected by this limitation.

For the compressed register file, the SIMTIGHT authors report that a $^1/_4$-size VRF provides a 68% storage reduction with negligible run-time and memory-access overheads. However, they were using a modern GCC (version 12) while we are using a relatively old Clang (the latest version that supports CHERI at the time), which yields inferior results. We therefore opt for a $^3/_8$-size VRF in the baseline providing a 55% storage reduction, as shown in Table 1.

We consider three main configurations of SIMTIGHT:

- **Baseline** A baseline configuration with a compressed general-purpose register file and a compressed stack cache, but without support for CHERI. Benchmarks run with no memory safety.
- **CHERI** An extension of the baseline configuration with CHERI. Value regularity in capability metadata is not detected or exploited. No CHERI instructions are implemented in

VRF Size (Vector Registers)	Total Storage (Kilobits)	Compression Ratio	Cycle Overhead	Main Memory Access Overhead
1,024 (1/2)	1,202	1 : 0.57	0.8%	0.1%
768 (3/8)	937	1 : 0.45	0.9%	2.2%
512 (1/4)	672	1 : 0.32	4.3%	39.9%

Table 1: Storage savings and geometric-mean overheads in the SIMTIGHT baseline, using Clang 13, due to register-file compression for a 1/2-size, 3/8-size, and 1/4-size VRF relative to an uncompressed register file (which contains 2,048 vector registers occupying 2,097 kilobits of storage).

the shared function unit. Benchmarks run with full spatial memory safety and referential integrity.

- **CHERI (Optimised)** An extension of the CHERI configuration with optimisations. Value regularity in capability metadata is detected and exploited. The capability-metadata register file is compressed. The shared VRF and null-value optimisations are both enabled. Capability metadata is cached in the compressed stack cache. CHERI instructions for getting and setting bounds are implemented in the shared function unit. This configuration also enables the static PC metadata restriction.

4.2 Threat Model

We consider a simple threat model in which an attacker seeks to exploit an out-of-bounds memory access to exfiltrate or corrupt data, or hijack control flow. When CHERI is enabled, applications are compiled in pure capability mode with all C++ pointers (and architectural addresses, such as the stack pointer and return addresses) being implemented as capabilities, enabling deterministic prevention of all such attacks. This covers code running on the SIMTIGHT SM and the host CPU, which can freely exchange capabilities via main memory. CHERI lays the foundation for protection against a much wider range of threats, including exploitation of use-after-free bugs, and interaction with untrusted software components. These threats are beyond the scope of this report, but we refer the reader to detailed security evaluations of CHERI on CPUs, which may be indicative [3, 27, 35, 38, 74].

4.3 Register-File Overhead

Figure 13 shows the proportion of register values that need be stored as uncompressed vectors in the shared VRF, both for values stored in the general-purpose register file and the capability-metadata register file. With the null-value optimisation, only the BlkStencil benchmark uses space in the VRF for capability metadata (explained below). The register-file storage overhead of CHERI is therefore almost entirely accounted for by the cost of the capability-metadata SRF, which is 14% of the total register-file storage of the baseline.

Figure 14 shows that no benchmark uses more than half of the available registers to hold capabilities. Therefore, with some basic compiler support to guarantee that only a subset of registers are used to store capabilities, the size of the capability-metadata SRF could be halved without impacting run-time performance in this benchmark suite. This would bring the register-file storage overhead of CHERI down to 7%.

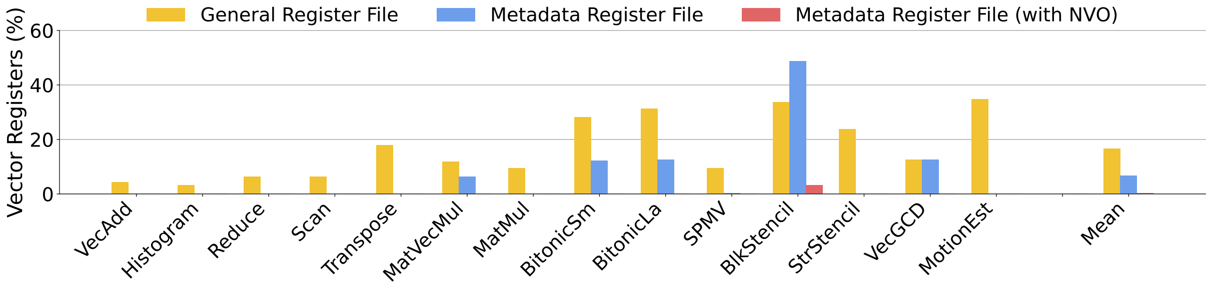


Figure 13: Proportion of registers stored as vectors in the VRF (lower is better) for the general-purpose register file, and for the capability-metadata register file with and without the null-value optimisation (NVO). Remaining registers are stored compactly as scalars in the SRF. Note that for the metadata register file, there are often no registers stored in the VRF, which is represented by the absence of a bar.

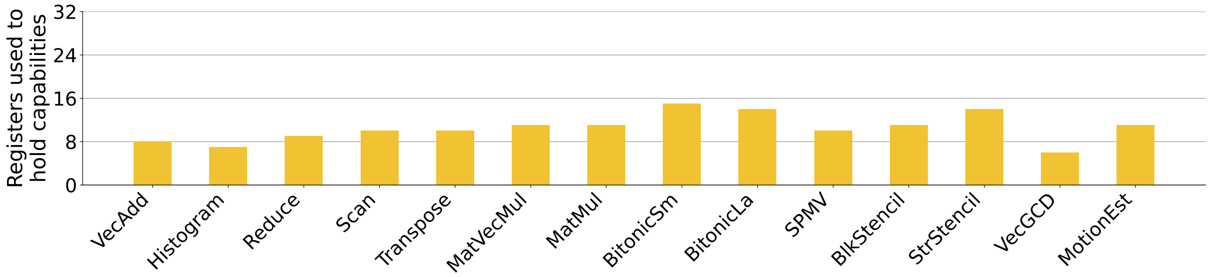


Figure 14: Number of registers per thread used to hold capabilities. Each thread has access to 32 registers in total. The remaining registers are never used to hold capabilities.

In the literature, the register file is typically considered to account for around 256KB of onchip storage per SM, compared to 64KB per SM for scratchpad memory and 64KB per SM for the L1 cache [28]. Furthermore, the shared L2 cache typically accounts for around 128KB to 256KB of storage per SM in modern GPGPUs [56]. We therefore estimate that the register file accounts for less than half of total onchip GPGPU storage. The total storage overhead of CHERI in a full GPGPU design would therefore likely fall below 3.5%.

Upon inspection of the BlkStencil benchmark, we see that the capability-metadata divergence arises from a compiler optimisation. A line of source code of the form

$$\texttt{if}\ (cond)\ \{\texttt{acc += }\star p_1\texttt{;}\}\ \texttt{else}\ \{\texttt{acc += }\star p_2\texttt{;}\}$$

effectively gets transformed to

$$\texttt{if}\ (cond)\ \{\texttt{tmp = }p_1\texttt{;}\}\ \texttt{else}\ \{\texttt{tmp = }p_2\texttt{;}\}\texttt{; acc += }\star\texttt{tmp;}$$

where p_1 and p_2 point to elements of different arrays (one stored in global memory and the other in shared local memory). The compiler has therefore transformed control-flow divergence into pointer-value divergence. In this work, we are using entirely pre-existing compiler toolchains but the above optimisation could potentially be disabled for SIMT targets, to preserve value regularity.

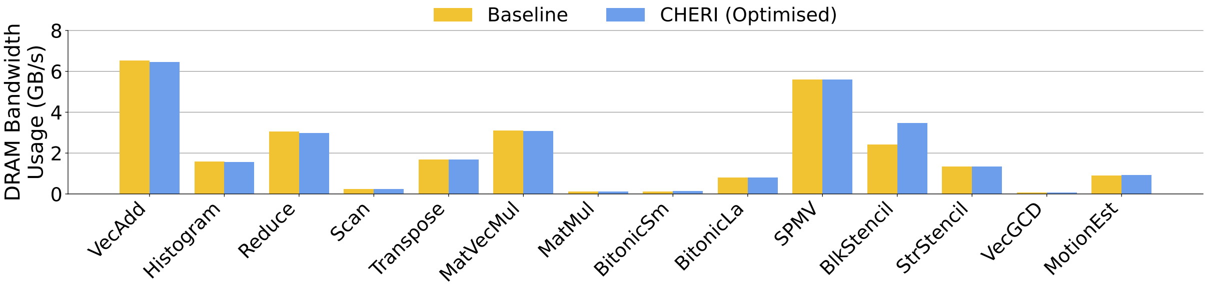


Figure 15: DRAM bandwidth usage (lower is better) with and without CHERI.

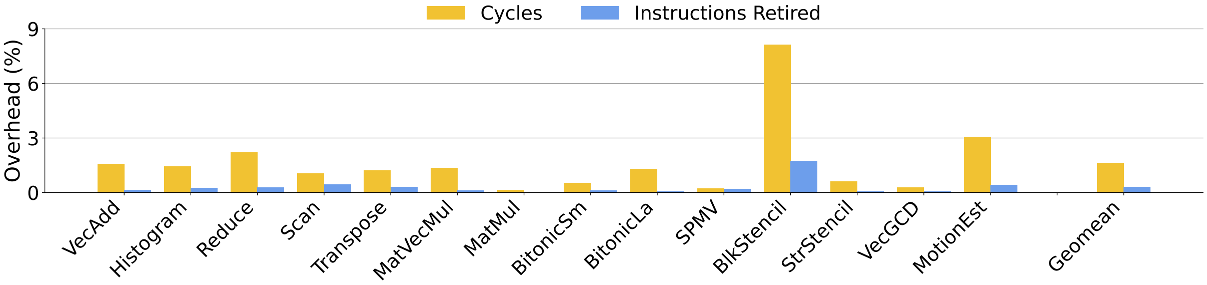


Figure 16: Execution-time overheads of the optimised CHERI configuration relative to the baseline configuration (lower is better), both in terms of clock cycles taken and instructions retired.

4.4 Memory-Bandwidth Overhead

Figure 15 shows that the introduction of CHERI does not significantly affect DRAM bandwidth usage in SIMTIGHT. Inlining of GPGPU functions plays an interesting role here. Without inlining, CHERI incurs a slightly larger overhead, which is partly ameliorated by the ability of SIMTIGHT's compressed stack cache to cache capability metadata. This is because function calls introduce more stack accesses, and hence more loading and storing of double-size pointers to memory. With aggressive inlining, caching of capability metadata does not appear to have a major impact on DRAM bandwidth, despite the potential to reduce the cost of register spilling.

4.5 Execution-Time Overhead

Figure 16 shows the execution-time overhead of adding CHERI to SIMTIGHT. The geometric-mean cycle overhead is 1.6%. We note that this is lower than CHERI overheads reported on CPU prototypes [68], suggesting that CHERI is cheaper on GPGPUs than CPUs. This is likely due to heavy function-call inlining and lack of pointer chasing in GPGPU workloads, limiting the number of accesses to double-sized pointers on the stack and heap respectively.

The main outlier in the execution-time results is the BlkStencil benchmark, which exhibits two uncommon but costly behaviours: capability-metadata divergence (Section 4.3) and execution of a relatively high number of CSC instructions, each of which can incur a one-cycle performance penalty during operand fetch in the SIMTIGHT microarchitecture (Section 3.2).

Configuration	**Area (ALMs)**	**Area (DSPs)**	**Block RAM (Kilobits)**	**Fmax (MHz)**
Baseline	126,753	0	2,156	180
CHERI	166,796	0	4,399	181
CHERI (Optimised)	149,356	0	2,394	180

Table 2: Synthesis results for a single SIMTIGHT SM on an Intel Stratix 10 FPGA with and without CHERI. The use of DSP blocks on the FPGA has been disabled to obtain a single ALM count representing all logic used.

4.6 Synthesis Results

Table 2 shows the logic area and onchip storage overheads of adding CHERI to SIMTIGHT. Our optimitisations reduce the area overhead by 44% to 708 ALMs per vector lane, comparable to (but slightly larger than) the cost of an additional multiplier (567 ALMs) per vector lane. The onchip memory storage overhead, measured in bits, is largely eliminated.

Compared to academic prototype implementations of CHERI on CPUs [67], our area overheads are relatively low; we have been able to amortise the cost of CHERI logic across multiple execution units. Furthermore, Arm's prototype implementation of CHERI in their Neoverse N1 CPU exhibits significantly lower area overheads [73] than academic CPU prototypes as the baseline design is richer in terms of architectural features and microarchitectural optimisations. For similar reasons, it is reasonable to expect that the relative area overheads of CHERI in a commercial-grade GPGPU would be lower than those for SIMTIGHT.

4.7 Software Bounds Checking

For comparison against CHERI, we have developed an experimental Rust port of NOCL and its benchmark suite that runs on the SIMTIGHT GPGPU. Rust is a modern general-purpose systems programming language that can serve as a suitable replacement for C/C++, providing similar levels of efficiency but with stronger correctness properties. In safe Rust, memory safety is enforced: all references point to valid live memory and cannot be accessed out of bounds.

Our Rust port is 'like-for-like' in the sense that C++ and Rust versions of each benchmark are defined very similarly. This gives confidence that different versions of the same benchmark are behaving in a similar way, allowing a fair performance comparison. On the other hand, it overlooks an important property of Rust. Rust's ownership system prohibits multiple threads from having write access to the same memory region at the same time, preventing data races. But such accesses are commonplace in data-parallel languages such as CUDA and OpenCL, e.g., multiple threads writing different parts of a result array at the same time. Resolving this discrepancy is an open problem, and we discuss some recent research in this area in Section 5.2. In our experimental NOCL port, we do not attempt to solve this issue; the Rust compiler is simply unaware of the presence of multiple GPU threads.

Figure 17 compares the execution times of C++ and Rust versions of the benchmarks running on SIMTIGHT. The C++ and Rust compilers used are both based on version 19.1.7 of LLVM. Overall, there is a geometric mean overhead of 46% from using Rust. If we disable bounds checks, this overhead drops to 9%. Bounds checking accounts for a 34% overhead in Rust alone. This indicates the software bounds checking is fairly expensive in low-level GPGPU

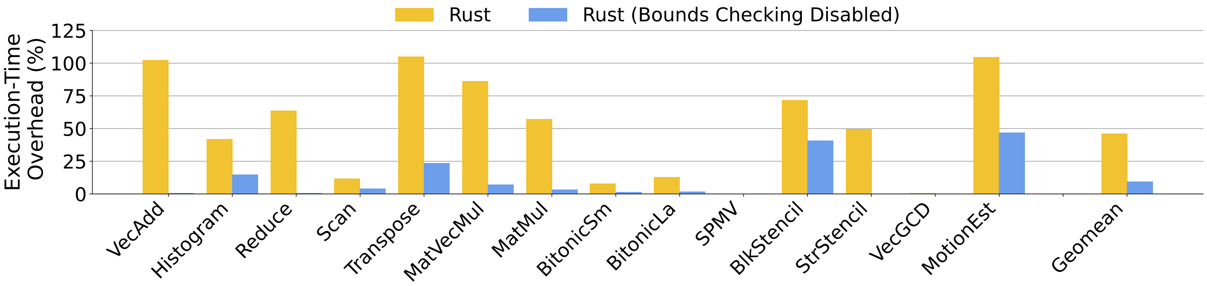


Figure 17: Execution-time overheads of our Rust port of NoCL running on SIMTight.

code, at least without providing further information about the relationships between the sizes of the buffers being accessed, which the compiler could use to eliminate some of the checks.

5 Related Work

5.1 GPU Debugging Tools

Memory-safety violations on GPUs are sufficiently common that a range of debugging tools have been built to detect them [22,24,55,62,69]. CUDA-MEMCHECK [55] and Oclgrind [62] tools can reliably detect buffer overflows (and other common errors) in CUDA and OpenCL code respectively, but introduce order-of-magnitude performance overheads. Canary-based approaches have much higher performance [22, 24]. They work by allocating padding bytes before/after each buffer and monitoring whether these bytes are ever modified, but they cannot detect out of bounds reads or access violations that fall outside the padding bytes. Very recently, NVIDIA has developed cuCatch [69] for probabilistically catching memory safety violations in CUDA code with a mean execution-time overhead of 19%. While useful, these tools are not appropriate from a security perspective where a viable solution will ideally provide both strong deterministic memory safety and low overheads.

5.2 Safe Languages Targeting GPUs

Over the past decade, GPUs have become a target for a number of high-level data-parallel array programming languages [25,30,32,34,47,59]. The goal is to move from low-level management of individual threads and memory accesses to high-level combinations of bulk parallel array operations such as map, reduce, scan, and so on. These languages have been shown to achieve useful levels of performance while often guaranteeing properties such as memory safety and data-race freedom. Interestingly, many bulk array operations are safe by construction and do not require bounds checks. However, some operations, such as gather and scatter, permit ad-hoc indexing and do require run-time safety checks. The authors of the high-level array language Futhark [34] report a 6% average performance overhead due to bounds checking on GPUs [33].

The flip side of abstracting away from low-level details is that the programmer loses control over these details. Köpcke et al. argue that such control is needed to extract the highest levels of performance from GPU hardware [40]. Inspired by Rust, they propose a safe low-level language for GPU programming called Descend. They use the concept of *views* to safely describe parallel accesses to shared memory in a way that can be statically checked for memory safety and data-race freedom. Essentially, this means that the programmer expresses both the individual memory accesses of each thread and a form of proof that these accesses are safe. As a result, Descend programs look quite different to CUDA programs and are harder to write; there are likely to be major costs in terms of porting existing applications and re-educating users. As in Rust, there are also cases where it is difficult to express the desired behaviour in a way that can be checked by the compiler, and Descend provides `unsafe` code blocks as an escape hatch. Any programs containing such blocks are no longer guaranteed to be safe.

Compiling Rust itself down to GPGPUs has also been explored. The Rust-CUDA project [20] extends the Rust compiler with support for targeting NVVM IR, a subset of LLVM IR. The CUDA toolkit ships with a closed-source library `libnvmm` that converts NVVM IR to NVIDIA's PTX portable bytecode format, which can in turn be compiled to GPGPU assembly by a closed-source NVIDIA device driver. The project is described as being in the early stages of development, providing two sample compute kernels written in Rust, but does not yet present any performance analysis. Currently, the sample compute kernels are declared as `unsafe`. Very recently, the developers have announced a new effort to pursue the project further [44].

Finally, it is worth noting that CHERI and safe languages are likely to be complementary on GPGPUs, just as they are considered to be on CPUs [10].

5.3 Hardware Support for GPU Memory Safety

To our knowledge, the only prior hardware approach to GPU memory safety is GPUShield [41]. This extends NVIDIA and Intel GPU models with a *bounds table* and uses the top 16 bits of every 64-bit pointer to hold an index into the table. The bounds table is setup before a GPU kernel is launched, and remains unchanged for the duration of execution. Bounds on kernel arguments are provided by the caller, and the compiler is extended to emit bounds information for buffers declared within a kernel. The compiler is also modified to avoid bounds checking on accesses that are known to be safe by static analysis; this is achieved by marking pointers as 'unprotected' in the top 16 bits.

The authors argue that extending the size of registers is unacceptable on GPUs due to memory bandwidth overheads and already-massive GPU register files. However, our results suggest that these costs can be largely eliminated by exploiting inter-thread redundancy in the metadata. Furthermore, extending the pointer size has a number of benefits, such as retaining access to the full address space, supporting 32-bit architectures as well as 64-bit architectures, and increasing the number of bits available to encode metadata. This in turn avoids the need for a limit on the number of buffers that can be protected, and indeed the need for indirection via a bounds table in the first place, as bounds can be encoded directly within the metadata itself.

GPUShield has been developed with a strong emphasis on efficiency, and the authors report a low average execution-time overhead of 0.8%. However, there are limitations in terms of expressibility and security. From an expressibility perspective, bounds cannot be modified during kernel execution, meaning that dynamically allocated memory cannot be protected. From a security perspective, pointer metadata is unprotected, which means that the buffer id can be corrupted or forged. Furthermore, GPUShield's 'unprotected' pointers, which allow the bounds table lookup to be bypassed for efficiency, makes it possible to forge a pointer to any address in memory. As GPUShield was developed in simulation, the area overhead is unclear,

Feature	**GPUShield**	**CHERI**
Supports spatial memory safety	✓	✓
Provides referential integrity	✗	✓
Supports both 32-bit and 64-bit architectures	✗	✓
Permits use of entire address space	✗	✓
Supports an unlimited number of buffers	✗	✓
Supports dynamic allocation of buffers on the GPU	✗	✓
Supports isolation of software components including GPU kernels	✗	✓
Allows pointers to be distinguished from data	✗	✓
Applies to both CPUs and GPUs	✗	✓
Demonstrated in a synthesisable GPU core	✗	✓
Run-time performance overhead on GPUs	Low	Low
Silicon area overhead on GPUs	Low (likely)	Moderate

Figure 18: Feature comparison of GPUShield [41] and CHERI.

but it would probably be lower than CHERI's as data-path widths are largely unaffected and bounds are uncompressed. A feature comparison of GPUShield and CHERI is summarised in Figure 18.

6 Conclusion

Despite huge numbers of registers in heavily-threaded GPGPU cores, extending registers with metadata can, in fact, be feasible. Storage costs in SIMT designs are dependent on the amount of value regularity between threads executing in lockstep. Capability metadata exhibits substantial value regularity, which can be exploited to largely eliminate the storage overhead of CHERI's double-size registers. Furthermore, the logic-area overhead of CHERI can be significantly reduced by amortising the cost of some CHERI instructions across multiple execution units. With these optimisations, CHERI offers a viable path to memory-safe GPGPU applications written in CUDA and OpenCL, avoiding the need for widespread porting of applications to new memory-safe languages and the associated re-education of users. Compared to state-of-the-art work on hardware support for GPU memory safety, CHERI provides far stronger security properties with similar runtime overheads, but likely higher area overheads. CHERI supports a much broader threat model, which in future would be interesting to explore in the context of GPGPUs.

Acknowledgements

Thanks to Jianyi Cheng for improving the reproducibility of the results presented in this report. This work was supported by the UK EPSRC under the *CAPcelerate Project* (EP/V000381/1) and the *Chrompartments Project* (EP/X015963/1), both part of the Digital Security by Design (DSbD) Programme and the DSbDtech initiative. CHERI support for SIMTight has been merged into the main SIMTight distribution, available at `https://github.com/CTSRD-CHERI/SIMTight`.

References

[1] Advanced Micro Devices (AMD). *Southern Islands Series Instruction Set Architecture 1.1*, 2012.

[2] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. FGPU: An SIMT-Architecture for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2016)*.

[3] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2023.

[4] Kevin Andryc, Murtaza Merchant, and Russell Tessier. FlexGrip: A soft GPGPU for FPGAs. In *International Conference on Field-Programmable Technology (FPT 2013)*.

[5] Arm. Arm Morello Program. https://www.arm.com/architecture/cpu/morello (accessed 2024-01-30), 2024.

[6] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimisation*, 12(2), 2015.

[7] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Research report, NVIDIA Corporation, 2008.

[8] Jeff Bush, Mohammad A. Khasawneh, Khaled Z. Mahmoud, and Timothy N. Miller. NyuziRaster: Optimizing rasterizer performance and energy in the Nyuzi open source GPU. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2016)*.

[9] Zhongliang Chen and David Kaeli. Balancing Scalar and Vector Execution on GPU Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*.

[10] David Chisnall. CHERI Myths: I don't need CHERI if I have safe languages. https://cheriot.org/cheri/myths/2024/08/28/cheri-myths-safe-languages.html (accessed 2024-12-19), 2024.

[11] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*.

[12] Chromium Team. The Chromium Projects: Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/ (accessed 2024-01-25), 2024.

[13] Cybersecurity & Infrastructure Security Agency (CISA), National Security Agency (NSA), Federal Bureau of Investigation (FBI), Australian Cyber Security Centre (ACSC), Canadian Centre for Cyber Security (CCCS), United Kingdom's National Cyber Security Centre (NCSC-UK), Germany's Federal Office for Information Security (BSI), Netherlands' National Cyber Security Centre (NCSC-NL), Computer Emergency Response Team New Zealand (CERT NZ), and New Zealand's National Cyber Security Centre (NCSC-NZ). Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default. https://www.cisa.gov/sites/default/files/2023-04/principles_approaches_for_security-by-design-default_508_0.pdf (accessed 2024-08-02), 2023.

[14] Cybersecurity & Infrastructure Security Agency (CISA), National Security Agency (NSA), Federal Bureau of Investigation (FBI), Australian Cyber Security Centre (ASD's ACSC), Canadian Centre for Cyber Security (CCCS), United Kingdom's National Cyber Security Centre (NCSC-UK), Computer Emergency Response Team New Zealand (CERT NZ), and New Zealand's National Cyber Security Centre (NCSC-NZ). The Case for Memory Safe Roadmaps: Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (accessed 2025-02-18), 2023.

[15] Codasip. Codasip delivers processor security to actively prevent the most common cyberattacks. https://codasip.com/press-release/2023/10/31/codasip-delivers-processor-security-to-actively-prevent-cyberattacks/ (accessed 2024-01-30), 2024.

[16] Caroline Collange. Simty: generalized SIMT execution on RISC-V. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*.

[17] Caroline Collange. Identifying scalar behavior in CUDA kernels. Research report, ENS Lyon, 2011.

[18] Caroline Collange. Stack-less SIMT reconvergence at low cost. Research report, ENS Lyon, 2011.

[19] Caroline Collange, David Defour, and Yao Zhang. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. In *Euro-Par 2009 – Parallel Processing Workshops*.

[20] Riccardo D'Ambrosio. The Rust CUDA Project. https://github.com/Rust-GPU/Rust-CUDA (commit 8a6cb73, accessed 2024-02-01), 2021.

[21] Bang Di, Jianhua Sun, and Hao Chen. A Study of Overflow Vulnerabilities on GPUs. In *International Conference on Network and Parallel Computing*, 2016.

[22] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. GMOD: a dynamic GPU memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2018, 2018.

[23] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for GPGPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[24] Christopher Erb and Joseph L. Greathouse. clARMOR: A Dynamic Buffer Overflow Detector for OpenCL Kernels. In *Proceedings of the International Workshop on OpenCL*, IWOCL 2018, 2018.

[25] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1), 2018.

[26] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[27] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2024.

[28] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *45th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[29] Syed Zohaib Gilani, Nam Sung Kim, and Michael Schulte. Power-efficient computing for compute-intensive GPGPU applications. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT 2012).*

[30] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2011.

[31] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Conference on Security Symposium*, 2024.

[32] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020.

[33] Troels Henriksen. Bounds Checking on GPU. *International Journal of Parallel Programming*, 49(6), 2021.

[34] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[35] Graeme Jenkinson, Alfredo Mazzinghi, and Robert N. M. Watson. CHERI-based memory protection and compartmentalisation for web services on Morello. Technical Report, Capabilities Limited, `https://www.capabilitieslimited.co.uk/_files/ugd/893621_985a92a599bf41208e4c5710abcf3a68.pdf` (accessed 2025-03-12), 2024.

[36] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking, 2019. arXiv 1903.07486.

[37] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. Efficient Tagged Memory. In *International Conference on Computer Design (ICCD)*, 2017.

[38] Nicolas Joly, Saif ElSherei, and Saar Amar. Security Analysis of CHERI ISA. Technical Report, Microsoft Security Response Center, `https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf` (commit 1372d4f, accessed 2025-03-12), 2020.

[39] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. Microarchitectural mechanisms to exploit value structure in SIMT architectures. In *40th Annual International Symposium on Computer Architecture (ISCA 2013)*.

[40] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. Descend: A Safe GPU Systems Programming Language. *Proceedings of the ACM on Programming Languages*, 8(PLDI), 2024.

[41] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing GPU via region-based bounds checking. In *49th International Symposium on Computer Architecture (ISCA)*, 2022.

[42] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-Compression: Enabling Power Efficient GPUs through Register Compression. In *42nd Annual International Symposium on Computer Architecture (ISCA 2015)*.

[43] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. Convergence and scalarization for data-parallel architectures. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2013)*.

[44] Christian Legnitto. Rebooting the Rust CUDA project. `https://rust-gpu.github.io/blog/2025/01/27/rust-cuda-reboot/` (accessed 2025-02-07), 2025.

[45] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2), 2008.

[46] Zhenhong Liu, Syed Gilani, Murali Annavaram, and Nam Sung Kim. G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA 2017)*.

[47] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[48] Andrea Miele. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12(2), 2016.

[49] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL (accessed 2024-01-25), 2019.

[50] Sparsh Mittal. A Survey of Techniques for Architecting and Managing GPU Register File. *IEEE Transactions on Parallel and Distributed Systems*, 28(1), 2017.

[51] Matthew Naylor. SIMTight Open-Source Repository. https://github.com/CTSRD-CHERI/SIMTight (commit 467bb3a, accessed 2024-12-20), 2024.

[52] Matthew Naylor. NoCL Manual. https://github.com/CTSRD-CHERI/SIMTight/blob/master/doc/NoCL.md (commit 46ad488, accessed 2025-02-18), 2025.

[53] Matthew Naylor, Alexandre Joannou, Paul Metzger, A. Theodore Markettos, Simon W. Moore, and Timothy M. Jones. Advanced Dynamic Scalarisation for RISC-V GPGPUs. In *42nd IEEE International Conference on Computer Design (ICCD)*, 2024.

[54] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[55] NVIDIA Corporation. CUDA-MEMCHECK User Manual. https://docs.nvidia.com/cuda/archive/11.7.1/pdf/CUDA_Memcheck.pdf (accessed 2023-02-24), 2022.

[56] NVIDIA Corporation. NVIDIA Ada GPU Architecture (v2.02). https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf (accessed 2025-01-07), 2023.

[57] NVIDIA Corporation. CUDA Code Samples. https://developer.nvidia.com/cuda-code-samples (accessed 2024-01-23), 2024.

[58] NVIDIA Corporation. NVIDIA OpenCL SDK Code Samples. https://developer.nvidia.com/opencl (accessed 2024-01-23), 2024.

[59] NVIDIA Corporation. Thrust, the CUDA C++ template library, Version 2.1.0. https://developer.nvidia.com/thrust (accessed 2024-12-18), 2024.

[60] NVIDIA Corporation. CUDA C++ Programming Guide (Release 12.8). https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (accessed 2025-02-13), 2025.

[61] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security*, 102(C), 2021.

[62] James Price and Simon McIntosh-Smith. Oclgrind: an extensible OpenCL device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL 2015, 2015.

[63] Alex Rebert, Ben Laurie, Murali Vijayaraghavan, and Alex Richardson. Google Security Blog: Securing tomorrow's software: the need for memory safety standards. https://security.googleblog.com/2025/02/securing-tomorrows-software-need-for.html (accessed 2025-02-26), 2025.

[64] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Murali Annavaram, and Woo Woo Ro. Improving Energy Efficiency of GPUs through Data Compression and Compressed Execution. *IEEE Transactions on Computers*, 66(05), 2017.

[65] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon Kim. Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023, Volume 3)*.

[66] Peter Rugg, Alexandre Joannou, Jonathan Woodruff, and Ivan Ribeiro. CheriCapLib Library. https://github.com/CTSRD-CHERI/cheri-cap-lib (commit 354a673, accessed 2024-10-30), 2024.

[67] Peter Rugg, Jonathan Woodruff, Alexandre Joannou, and Simon W. Moore. A Suite of Processors to Explore CHERI-RISC-V Microarchitecture. In *27th Euromico Digital System Design Conference (DSD)*, 2024.

[68] Peter David Rugg. Efficient spatial and temporal safety for microcontrollers and application-class processors. Technical Report UCAM-CL-TR-984, University of Cambridge, Computer Laboratory, July 2023.

[69] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 2023.

[70] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2021)*.

[71] Jeff Vander Stoep, Android Security & Privacy Team, Chong Zhang, and Android Media Team. Google Security Blog: Queue the Hardening Enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (accessed 2024-01-25), 2019.

[72] Kai Wang and Calvin Lin. Decoupled affine computation for SIMT GPUs. In *ACM/IEEE 44th International Symposium on Computer Architecture (ISCA 2017)*.

[73] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, September 2023.

[74] Robert N. M. Watson, Ben Laurie, and Alex Richardson. Assessing the Viability of an OpenSource CHERI Desktop Software Ecosystem. Technical Report, Capabilities Limited, https://www.capabilitieslimited.co.uk/_files/ugd/f4d681_e0f23245dace466297f20a0dbd22d371.pdf (accessed 2025-03-12), 2021.

[75] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. An Introduction to CHERI. University of Cambridge Technical Report, UCAM-CL-TR-941, 2019.

[76] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, 2023.

[77] Robert N.M. Watson, John Baldwin, David Chisnall, Tony Chen, Jessica Clarke, Brooks Davis, Nathaniel Filardo, Brett Gutstein, Graeme Jenkinson, Ben Laurie, Alfredo Mazzinghi, Simon Moore, Peter G. Neumann, Hamed Okhravi, Alex Richardson, Alex Rebert, Peter Sewell, Laurence Tratt, Murali Vijayaraghavan, Hugo Vincent, and Konrad Witaszczyk. It is time to standardize principles and practices for software memory safety. *Communications of the ACM*, 68(2), 2025.

[78] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[79] Jonathan Woodruff. *CHERI: a RISC capability machine for practical memory safety*. PhD Thesis, UCAM-CL-TR-858, University of Cambridge, 2014.

[80] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10), 2019.

[81] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *27th ACM International Conference on Supercomputing (ICS 2013)*.

[82] Ayse Yilmazer, Zhongliang Chen, and David Kaeli. Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs. In *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*.

Appendix A: Sample NoCL Benchmark

This appendix contains the full C++ source code for the Histogram benchmark written using the NoCL library, including both host-side code and device-side code. The host-side code allocates and initialises buffers, launches the kernel on the device, and verifies the output. It runs on the CPU in the SIMTight evaluation SoC. Meanwhile, the device-side code runs on the streaming multiprocessor in the SIMTight evaluation SoC. No modifications are required when compiling using the CHERI compiler.

```
#include <NoCL.h>
#include <Rand.h>

// Kernel for computing 256-bin histograms
struct Histogram : Kernel {
  // Parameters
  int len; unsigned char* in; int* out;

  // Histogram bins in shared local memory
  int* bins;

  void init() {
    declareShared(&bins, 256);
  }

  void kernel() {
    // Initialise bins
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
      bins[i] = 0;

    __syncthreads();

    // Update bins
    for (int i = threadIdx.x; i < len; i += blockDim.x)
      atomicAdd(&bins[in[i]], 1);

    __syncthreads();

    // Write bins to global memory
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
      out[i] = bins[i];
  }
};


int main()
{
  // Are we in simulation?
  bool isSim = getchar();

  // Vector size for benchmarking
  int N = isSim ? 3000 : 1000000;

  // Input and output vectors
  nocl_aligned unsigned char input[N];
  nocl_aligned int output[256];

  // Initialise inputs
  uint32_t seed = 1;
  for (int i = 0; i < N; i++)
    input[i] = rand15(&seed) & 0xff;

  // Instantiate kernel
  Histogram k;

  // Use single block of threads
  k.blockDim.x = SIMTLanes * SIMTWarps;

  // Assign parameters
  k.len = N;
  k.in = input;
  k.out = output;

  // Invoke kernel
  noclRunKernelAndDumpStats(&k);

  // Check result
  bool ok = true;
  int goldenBins[256];
  for (int i = 0; i < 256; i++) goldenBins[i] = 0;
  for (int i = 0; i < N; i++) goldenBins[input[i]]++;
  for (int i = 0; i < 256; i++)
    ok = ok && output[i] == goldenBins[i];

  // Display result
  puts("Self␣test:␣");
  puts(ok ? "PASSED" : "FAILED");
  putchar(’\n’);

  return 0;
}
```

Appendix B: Reproducing the Results

This appendix contains instructions for reproducing the main results of the paper. These instructions have been tested on Ubuntu 20.04 with Docker 24.0.7, but should work on any system with a suitable Docker installation.

To begin, recursively clone the `cheri-report` branch of the SIMTight repository:

```
git clone -b cheri-report --recursive https://github.com/CTSRD-CHERI/SIMTight
```

To satisfy the dependencies of the project, simply enter a Docker shell:

```
cd SIMTight && make shell
```

This takes around 7 minutes in the first instance, but only a matter of seconds thereafter.

Download and build the exact version of the CHERI compiler used in this report, and add it to the environment:

```
cd cheri-tools && ./build-cheri.sh
cd cheri-tools && source ./add-cheri-tools-to-path.sh
```

This takes around 25 minutes to complete. Note that if you exit the Docker shell, and later re-enter it, you will not need to rebuild the CHERI tools but you will need to re-add them to the environment.

To enable CHERI support in SIMTight, modify `inc/Config.h` to contain:

- `#define EnableTaggedMem 1`
- `#define EnableCHERI 1`
- `#define UseClang 1`

The SIMTight evaluation SoC can then be built and tested in simulation with the command:

```
cd test && ./test.sh
```

This takes around 90 minutes to complete. It runs the RISC-V test suite on both the CPU and the SM. It also runs all the NoCL benchmarks, which are self-checking. The expected output is:

```
SIMTight build: ok
Simulator build: ok
Starting simulator: ok

Test Suite (CPU, Simulation)
============================

  I/add                  ok
  I/addi                 ok
  I/and                  ok
  I/andi                 ok
  I/beq                  ok
  I/bge                  ok
  I/bgeu                 ok
  I/blt                  ok
  I/bltu                 ok
  I/bne                  ok
  I/lui                  ok
  I/or                   ok
  I/ori                  ok
  I/simple               ok
  I/sll                  ok
  I/slli                 ok
  I/slt                  ok
  I/slti                 ok
  I/sltiu                ok
  I/sltu                 ok
  I/sra                  ok
  I/srai                 ok
  I/srl                   ok
  I/srli                  ok
  I/sub                   ok
  I/xor                   ok
  I/xori                  ok
  M/div                   ok
  M/divu                  ok
  M/mul                   ok
  M/mulh                  ok
  M/mulhsu                ok
  M/mulhu                 ok
  M/rem                   ok
  M/remu                  ok
  CHERI/auipcc            ok
  CHERI/candperm          ok
  CHERI/ccleartag         ok
  CHERI/cgetaddr          ok
  CHERI/cgetbase          ok
  CHERI/cgetflags         ok
  CHERI/cgetlen           ok
  CHERI/cgetperm          ok
  CHERI/cgetsealed        ok
  CHERI/cgettag           ok
  CHERI/cgettype          ok
  CHERI/cincoffset        ok
  CHERI/cincoffsetimm     ok
  CHERI/cjalr             ok
  CHERI/clb               ok
  CHERI/clh               ok
  CHERI/clw               ok
  CHERI/cmove             ok
  CHERI/csb               ok
  CHERI/csc               ok
  CHERI/csealentry        ok
  CHERI/csetaddr          ok
  CHERI/csetbounds        ok
  CHERI/csetboundsexact   ok
  CHERI/csetboundsimm     ok
  CHERI/csetflags         ok
  CHERI/csh               ok
  CHERI/csub              ok
  CHERI/csw               ok

Summary: ok

Test Suite (SIMT Core, Simulation)
==================================

  I/add                   ok
  I/addi                  ok
  I/and                   ok
  I/andi                  ok
  I/beq                   ok
  I/bge                   ok
  I/bgeu                  ok
```

```
I/blt                   ok
I/bltu                  ok
I/bne                   ok
I/lui                   ok
I/or                    ok
I/ori                   ok
I/simple                ok
I/sll                   ok
I/slli                  ok
I/slt                   ok
I/slti                  ok
I/sltiu                 ok
I/sltu                  ok
I/sra                   ok
I/srai                  ok
I/srl                   ok
I/srli                  ok
I/sub                   ok
I/xor                   ok
I/xori                  ok
M/div                   ok
M/divu                  ok
M/mul                   ok
M/mulh                  ok
M/mulhsu                ok
M/mulhu                 ok
M/rem                   ok
M/remu                  ok
CHERI/auipcc            ok
CHERI/candperm          ok
CHERI/ccleartag         ok
CHERI/cgetaddr          ok
CHERI/cgetbase          ok
CHERI/cgetflags         ok
CHERI/cgetlen           ok
CHERI/cgetperm          ok
CHERI/cgetsealed        ok
CHERI/cgettag           ok
CHERI/cgettype          ok
CHERI/cincoffset        ok
CHERI/cincoffsetimm     ok
CHERI/cjalr             ok
CHERI/clb               ok
CHERI/clh               ok
CHERI/clw               ok
CHERI/cmove             ok
CHERI/csb               ok
CHERI/csc               ok
CHERI/csealentry        ok
CHERI/csetaddr          ok
CHERI/csetbounds        ok
CHERI/csetboundsexact   ok
CHERI/csetboundsimm     ok
CHERI/csetflags         ok
CHERI/csh               ok
```

```
  CHERI/csub              ok
  CHERI/csw               ok
  CHERI/A/camoadd_w       ok
  CHERI/A/camoand_w       ok
  CHERI/A/camomax_w       ok
  CHERI/A/camomaxu_w      ok
  CHERI/A/camomin_w       ok
  CHERI/A/camominu_w      ok
  CHERI/A/camoor_w        ok
  CHERI/A/camoswap_w      ok
  CHERI/A/camoxor_w       ok

Summary: ok

Apps (Simulation)
=================

Samples/VecAdd (build): ok
Samples/VecAdd (run): ok
Samples/Histogram (build): ok
Samples/Histogram (run): ok
Samples/Reduce (build): ok
Samples/Reduce (run): ok
Samples/Scan (build): ok
Samples/Scan (run): ok
Samples/Transpose (build): ok
Samples/Transpose (run): ok
Samples/MatVecMul (build): ok
Samples/MatVecMul (run): ok
Samples/MatMul (build): ok
Samples/MatMul (run): ok
Samples/BitonicSortSmall (build): ok
Samples/BitonicSortSmall (run): ok
Samples/BitonicSortLarge (build): ok
Samples/BitonicSortLarge (run): ok
Samples/SparseMatVecMul (build): ok
Samples/SparseMatVecMul (run): ok
InHouse/BlockedStencil (build): ok
InHouse/BlockedStencil (run): ok
InHouse/StripedStencil (build): ok
InHouse/StripedStencil (run): ok
InHouse/VecGCD (build): ok
InHouse/VecGCD (run): ok
InHouse/MotionEst (build): ok
InHouse/MotionEst (run): ok

All tests passed
```

In simulation, all benchmarks are run on small datasets to ensure timely completion. To obtain performance counters for each benchmark, the `--stats` option may be passed to `test.sh` (though these counters may not be particularly meaningful in simulation due to small datasets not masking startup costs).

The three main configurations of SimTight used in this report — **Basline**, **CHERI**, and **CHERI (Optimised)** — are defined in `scripts/sweep.py` and can be tested as follows.

```
cd scripts && ./sweep.py test
```

This will take around 5 hours to complete. Test results for the three configurations are written to the file `test/test.log`.

To obtain results on FPGA, we require:

- Version 22.1pro of Quartus in the path, with the environment variable `QUARTUS_ROOTDIR` pointing to it, and the environment variable `LM_LICENSE_FILE` pointing to a valid license.
- A Terasic DE10-Pro development board (revision D) connected via USB and visible as the sole device when running the `jtagconfig` command.

To build an FPGA image for SIMTIGHT:

```
cd de10-pro && make
```

This takes around an hour to complete.

To download the image onto the FPGA:

```
cd de10-pro && make download-sof
```

This takes around 30 seconds to complete.

To run all benchmarks on FPGA and obtain all performance counters:

```
cd test && ./test.sh --stats --fpga-d --apps-only
```

This takes around a minute to complete, assuming that the FPGA image has already been built (the script will build and download the FPGA image if it has not already been done, but a stepwise approach is more advisable when exercising the flow for the first time).

To reproduce the results for the three main configurations of SIMTIGHT used in the paper:

```
cd scripts && ./sweep.py bench
```

This takes around three hours to complete, as each configuration must be synthesised from scratch. The output is three `.bench` files in the `test` directory, one for each configuration. A `.bench` file is simply a file containing the output of `test.sh` with the `--stats` option enabled.

For FPGA synthesis results, we use Quartus Design Space Explorer to synthesise each configuration of SIMTIGHT across 16 different seeds, selecting the design with the highest Fmax. This long process can be initiated with the command:

```
cd scripts && ./sweep.py synth
```

This should be run on a modern server with at least 256GB of RAM, where it takes around a day to complete. A summary of the synthesis results is written to `de10-pro/synth.log`.