Implementing Hardware Extensions for Multicore RISC-V GPUs

Tine Blaise
Georgia Institute of Technology
Atlanta, Georgia
blaisetine@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, Georgia
hyesoon@cc.gatech.edu

ABSTRACT

As silicon technology scaling is approaching its limits, the semiconductor industry has adopted hardware specialization as a workaround to continue improving the performance of processors, creating new brands of multi-core processor architectures with specialized execution units and fixed-function hardware. For instance, modern GPUs today have dedicated custom hardware for image processing, 3D graphics, graph analytics, and machine learning acceleration.

In recent years, the RISC-V ISA adoption has increased, and several implementations of GPUs based on the RISC-V ISA were introduced that integrate multiple cores [5] [12] [6]. Extending a RISC-V-based GPU to support custom hardware acceleration while still maintaining compatibility with the RISC-V ISA is not a trivial task. Part of the challenge involves extending the instruction set and register file, but the other part is to figure out how the hardware addition will interface with the existing processor pipeline.

In this work, we present a generalized methodology for implementing hardware extensions for multi-core RISC-V-based GPUs. We discuss the various hardware extension architectures on GPUs and propose possible implementations on RISC-V. Our generalized solution addresses both the ISA and microarchitecture changes. We also provide a generalized solution for supporting hardware performance monitoring counters for platforms with multiple custom accelerators onboard. We showcase some applications of our methodology with a custom hardware extension implementation on RISC-V-based GPUs.

KEYWORDS

High-Performance Computing, multi-threading, heterogeneous Computing, Parallel Programming, Machine Learning, Systolic Array, Graphics, Performance Monitoring

ACM Reference Format:
Tine Blaise and Hyesoon Kim. 2022. Implementing Hardware Extensions for Multicore RISC-V GPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As silicon technology scaling is approaching its limits [7], the semiconductor industry has adopted multi-core architectures as a workaround to continue improving the performance of processors, creating accelerators with hundredth of processing cores [13]. Another trend has been the push for specialization with dedicated custom execution units and fixed-function hardware to accelerate specific operations for a target application domain. For instance, modern GPUs today implement dedicated custom hardware for image processing, 3D graphics, graph analytics, and machine learning acceleration [15] [13] [15] [17].

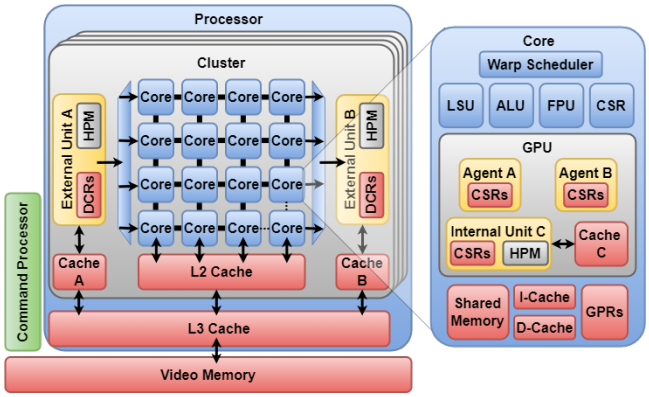

Figure 1: GPU Pipeline With Hardware Extension Units

In recent years, the RISC-V ISA adoption has increased, and several implementations of GPUs based on the RISC-V ISA were introduced that integrate multiple cores [5] [12] [6]. Most of these processors implement some of the RISC-V standard extensions, mainly integer multiply/divide and floating-point arithmetic. Implementing the RISC-V standard extensions is well documented in the specifications [21], which simplifies the task for the average developers. When it comes to non-standard extensions, such as adding special function units or supporting a matrix multiplication unit, for instance, there is no defined methodology to approach the problem. Extending a RISC-V-based GPU to support custom hardware while still maintaining compatibility with the RISC-V ISA is not a trivial task. Part of the challenge involves extending the instruction set, and register file as the new hardware requirements may not be compatible with the RISC-V ISA; the other part is to figure out how the hardware addition will interface with the existing processor pipeline.

In this work, we present a generalized methodology for implementing hardware extensions for multi-core RISC-V-based GPUs. Figure 1 illustrates an overview of a standard GPU pipeline with custom fixed-function units A, B, and C (see yellow blocks). Hardware fixed-function units A and B are located outside the GPU core and are shared by all cores within their enclosing cluster. Hardware fixed-function unit C is located inside each core, similar to most

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

RISC-V standard extensions and has direct access to the pipeline. The location of the custom accelerator block, whether it is upstream of the computer core or downstream, whether it is configurable or not, and whether it directly accesses memory or not, are some of the potential design requirements that will impact the final implementation. Most of the techniques introduced in this paper also apply to standard RISC-V-based single-core and multi-core CPUs.

We discuss the various hardware extension architectures on GPUs and propose possible implementations on RISC-V. Our generalized solution addresses both the ISA and microarchitecture changes. We also provide a generalized solution for supporting hardware performance monitoring counters for platforms with multiple custom accelerators onboard. We showcase some applications of our methodology with custom hardware extension implementations on RISC-V-based GPUs.

This paper makes the following key contributions:

- We propose a topology for GPU hardware extensions in term of their design requirements and interaction with the GPU pipeline.
- We present a generalized methodology for implementing hardware extensions for multi-core RISC-V-based GPUs.
- We provide a generalized solution for supporting hardware performance monitoring counters for platforms with multiple fixed-function units onboard.
- We showcase some applications of our methodology with a custom hardware extension implemented on RISC-V-based Vortex GPU [20].

2 BACKGROUND

2.1 Processor Hardware Extensions

Processors have been implementing custom fixed-function hardware and have exposed their access via their ISA extension since their inception. GPUs have also followed this approach to accelerate compute-intensive operations.

2.1.1 CPU Hardware Extensions. The common hardware extensions implemented into today's commodity CPUs [16] [14] [19] [18] include floating-point units, cryptography instructions [22] [9], Fused-multiply-add units, random number generators, Hamming weight computation, special instructions for neural network computation [2], etc.

2.1.2 GPU Hardware Extensions. On the graphics processing units, a large portion of the hardware extension is dedicated to the graphic acceleration and are implemented as fixed-function units, including rasterizer units, texture units, tesselation units, interpolation units, render output units, etc. Together with the standard hardware extensions such as floating-point units, modern GPUs also implement custom hardware for machine learning acceleration [4].

2.2 RISC-V ISA Extension

The RISC-V ISA specifications [**?**] define a base instruction set (RV32I) for integer arithmetic that is required by all implementations. The specifications also define optional standard extensions with their detailed functions and description. For supporting non-standard extensions, the specifications provide some facilities.

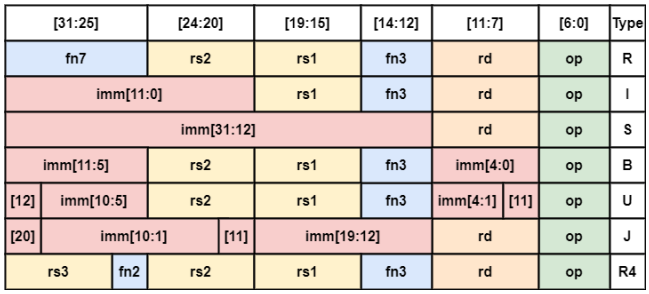

Figure 2: RISC-V 32-bit Instruction Formats

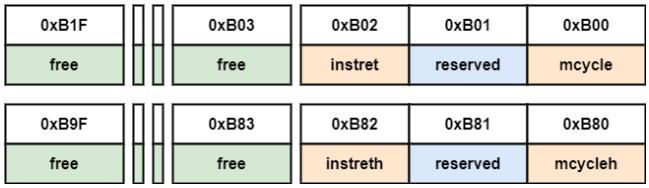

Figure 3: RISC-V Machine Performance Monitoring Counters

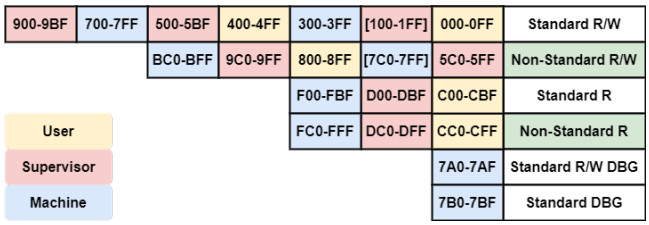

Figure 4: RISC-V CSRs Address Mapping

2.2.1 The Standard Extensions. The RISC-V ISA defines 15 standard extensions (M, A, F, D, Q, L, C, B, J, T, P, V, N, H, S) that provide acceleration for various operations, including integer multiplication, atomics, floating-point, vector operations, etc.

2.2.2 User-Defined Extensions. The RISC-V specifications reserve four custom opcodes **0x0B**, **0x2B**, **0x5B**, and **0x7B** for supporting user-defined extensions. Designers can use either one of these custom opcodes to define their custom instructions using the standard formats as described in Figure 3. The RISC-V instruction format only supports up to two source operands for integer operations and three operands for floating-point operations (R4), which is not always sufficient for some extensions; we will discuss workaround solutions in later sections.

2.2.3 Hardware Performance Monitoring Counters. Hardware performance monitoring counters (HPMs) are essential components of hardware evaluation as they enable the gathering of internal performance statistics. The RISC-V ISA defines 32 pairs of control status registers (CSRs) for storing HPMs where the lower half covers addresses **0xB00** to **0xB1F** and the upper half addresses **0xB80** to **0xB9F**. Three of the 32 registers are reserved for pre-defined counters that include the execution cycles and the number of instructions (see Figure 2), leaving only 27 free slots. In a platform

with multiple hardware extensions, we will propose a mapping to increase that number while still preserving application compatibility in later sections.

2.3 Vortex GPU Framework

The Vortex GPU [20] is a hardware implementation of a RISC-V-based SIMT multi-core processor with an integrated software and simulation task for architecture research. The GPU design has been optimized to FPGAs, peaking above 250 Mhz, and capable of fitting up to 64 cores (1024 threads) on Intel Stratix 10 FPGA [8]. These features make Vortex a safe choice for experimenting our extension experiments.

2.3.1 Vortex ISA. Vortex extends the standard RISC-V ISA with five new instructions *wspawn*, *tmc*, *split*, *join*, and *bar*, to support SIMT execution model and a *tex* instruction to accelerate texture sampling.

2.3.2 Software Stack. Vortex's flexible software stack allows for the execution of OpenCL programs. OpenCL is supported through a modified POCL [10] compiler backend to support RISC-V and Vortex's ISA extension [3].

2.3.3 Hardware Stack. Each processing core in Vortex implements a standard RISC-V five-stage pipeline with additional hardware blocks to support SIMT execution model. The execute unit integrates RISC-V standard extensions for integer multiplication and floating-point units. It also hosts a fixed-function texture accelerator block.

3 A TOPOLOGY OF HARDWARE EXTENSIONS

We propose a characterization of hardware extensions that will infer essential design decisions for the integration with the target processor pipeline. Table 1 summarizes a classification of common GPU hardware extensions. Our selected list contains a floating-point unit (FPU), fixed-point integer multiply-add (IMADD), SHA256 sum transformation function (SHA256Sum), matrix multiplication (MatMul), software prefetching (Prefetch), graphics vertex fetch engine (VFetch), graphics rasterizer (Raster), graphics attributes interpolation (Interp), texture sampling (Tex), graphics alpha blend (Blend). Table 2 provides the instruction operands signature for the selected hardware extensions. The MatMul instruction's operands are used to index on-chip matrix register files a, b, c where a and b are the source matrix indices and c is the destination. This implementation assumes another pair of custom instructions for loading and storing matrix elements to memory.

3.1 Producer vs Consumer Extensions

The role type column distinguishes extensions that consume their inputs from the processing cores, the consumers, and extensions that produce their output to the processing cores, the producers. The majority of hardware extensions generally operate as consumers and their execution is triggered by the processing core pipeline. Not all consumer extensions return a value, a software prefetch instruction (Prefetch) for instance will not have a return value. The graphics blend (Blend) extension takes pixel color information and write/blend the value into a destination pixel memory location. The GPU rasterizer (Raster) is a typical example of producer fixed-function hardware that generates pixel stamps for the processing core to operate on. Another graphics-related fixed-function unit that operates like a producer is the vertex fetch unit (VFetch) that accesses memory to process the vertex attributes needed as inputs to the processing cores. A producer extension always needs to produce a value for the core and notify the core when there is no data to process.

3.2 Internal vs External Extensions

The location of the hardware extension is also an important distinguishing factor. In a multi-core processor, an accelerator block can be placed inside the core or outside the core depending on the area resource constraints or invocation bandwidth. The default option is typically to keep the extension inside the core. This presents several advantages, 1) having no sharing between cores maximizes the compute bandwidth, which is ideal when the extension doesn't access memory, and 2) reduces design complexity as the new module can integrate with the existing interface used by other execute units. The main drawback of internal extensions is the area overhead since each core in the processor will be instantiating the new unit. And effective method to decide which location to place the hardware extension is to profile the application and determine how much bandwidth is needed. For instance, the graphics blend fixed-function (Blend) unit can reside inside or outside the core. However, pixel shader programs call the instruction once at the end of each iteration, and a single iteration generation typically executes hundredth and sometimes thousands of instructions. Placing the blend unit inside the core will add negligible performance improvement. On the contrary, the texture sampler unit (Tex) is preferably kept inside the core because typical pixel shader programs produce multiple invocations of the instruction per iteration.

3.3 Needing local storage

Some hardware extensions, depending on the implementation, may require some local storage to keep temporary data. An extension like matrix multiplication (MatMul) will need on-chip temporary storage to keep the elements of the source and destination matrices. This implementation is well suited for a scalar processor where wide vector register files are not available. In the graphics pipeline, the vertex fetch and rasterizer units also maintain on-chip storage for the attributes and pixel stamps, respectively, that they generate for the processing core to consume.

3.4 Accessing Memory

Hardware extensions that require access memory are typically located outside the core (have external location). This is mainly because of the memory latency, which requires queueing the pending requests from the core. These components are generally memory-bounds, which reduces the benefits of keeping the module local to the core. In practice, the hardware extension is coupled with a private cache that leverages its access pattern to improve data locality therein reducing the memory latency. The texture sampler unit in table 1 accesses memory while still remaining internal to the core because it is typically attached to a read-only cache that provides descent data locality, reducing the memory latency.

Name	Role Type	Location	Local Storage	Access Memory	Operands	Output Result	Configurable
FMAdd	Consumer	Internal	No	No	Standard	Yes	No
IMAdd	Consumer	Internal	No	No	Complex	Yes	No
SHA256Sum	Consumer	Internal	No	No	Standard	Yes	No
MatMul	Consumer	External	Yes	No	Standard	Yes	No
Prefetch	Consumer	Internal	Yes	Yes	Standard	No	No
VFetch	Producer	External	Yes	Yes	Standard	Yes	Yes
Raster	Producer	External	Yes	Yes	Standard	Yes	Yes
Interp	Consumer	Internal	No	No	Complex	Yes	Yes
Tex	Consumer	Internal	No	Yes	Standard	Yes	Yes
Blend	Consumer	External	No	Yes	Complex	No	Yes

Table 1: A Classification of Common GPU Hardware Extensions

Name	Output	Source Operands
FMAdd	result	a, b, c
IMAdd	result	a, b, c, shift
SHA256Sum	result	inb
MatMul	-	a, b, c
Prefetch	-	addr
VFetch	vertices	-
Raster	pixels	-
Interp	result	dx, dy, a, b, c
Tex	color	u, v, lod,
Blend	-	x, y, color, id, mask

Table 2: Hardware Extension Signatures

3.5 Complex Operands

Another design factor in implementing custom instructions is figuring out how to pass the inputs to the accelerator. Custom instructions that require more source operands than the limit defined by the RISC-V ISA which is three (see Figure 3) are classified as having complex operands. The graphic interpolation instruction (Interp) for instance needs five operands to compute the equation *f(dx, dy, a, b, c) = (dx * a) + (dy * b) + c*. Solving the challenge of handling complex operands will have some performance implications. We propose several strategies in the following section.

3.6 Configurable Extensions

Some extensions require configuration prior execution to change the operating function or provide constant parameters to the accelerator. On a typical GPU all the fixed-function components of the graphics pipeline export a configuration interface to feed constant parameters or modify their runtime behavior. The recommended method for configuring the hardware in RISC-V is to use control status registers (CSRs). Figure 4 shows the RISC-V CSRs address mapping across the different user, supervisor, and machine domains. The RISC-V specification has reserved unused CSRs slots for custom implementations, they fall under the non-standard CSR address space where 512 slots are read/write and 192 slots are read-only.

4 RISC-V ISA EXTENSION

This section discusses the various methods for extending the RISC-V ISA at the instruction level.

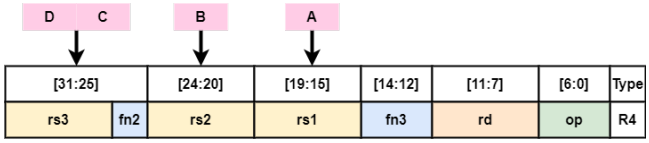

Figure 5: Operand Extension via Inputs Merging

4.1 Instruction Encoding

To encode the new instruction for a hardware extension. The developer can choose among the four available custom opcodes **0x0B**, **0x2B**, **0x5B**, and **0x7B** defined in the RISC-V specifications. The main challenge is choosing an adequate instruction format. This is the case because there are only four opcodes, and it is best to reuse them as much as possible if there is a desire to support future extensions. Figure 3 lists the most common formats with their corresponding encoding scheme. The commonly used formats are **R** and **R4**.

4.1.1 R format. is the most flexible format because 1) its operands' mix fits many hardware extensions that obey the generic **C = f(A, B)** signature. 2) this scheme encodes 3 bits into *func3* and 7 bits into *func7*, which provides a total of 1024 possible instruction definitions under that format.

4.1.2 R4 format. provides the best options when the number of operands on the extension is large, taking advantage of the third source operand in the format. The downside of this format is the limited options available to encode multiple instructions under that scheme. The format only has 5 bits (*func3* + *func2*) available to define 32 possible instructions.

In practice, implementations should prioritize R4 format and assign it at least two opcodes. This is because GPU's hardware extensions tend to require more than just two operands.

4.2 Operands Extension

GPU's Hardware extensions tend to require more than three operands and deciding how to pass the other source parameters to the accelerator can have significant performance implications. We experimented with three solutions for handling extra operands when implementing a custom instruction.

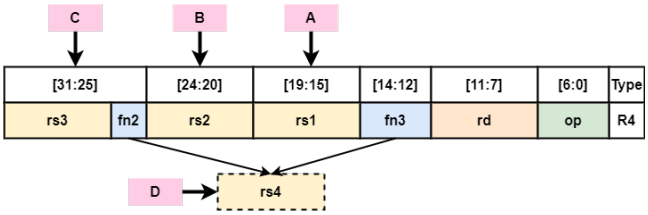

Figure 6: Operand Extension via Functions Merging

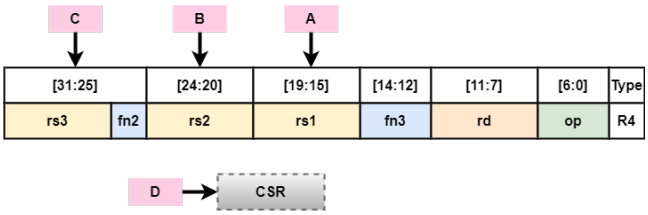

Figure 7: Operand Extension via CSRs

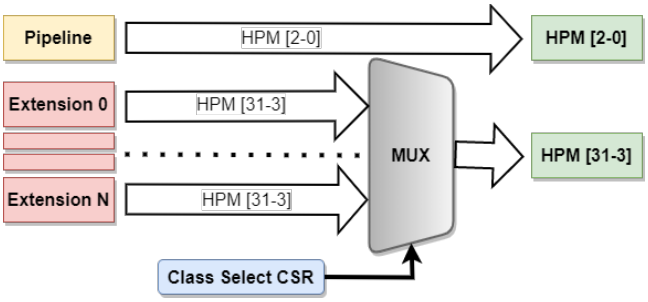

Figure 8: Hardware Performance Monitoring Extension

4.2.1 Inputs Merging. The most efficient solution to extend the operands is to simply merge the input arguments of the instruction to share the registers (see Figure 5). This is usually done by implementing an intrinsic function that wraps the native instruction and applies the merge. We found this method to be very effective for some graphics extensions like **Blend** where some arguments don't are guaranteed to be clamped within a certain range and maybe concatenated with others to share the same register.

4.2.2 Functions Merging. When the inputs arguments cannot be merged, the next option is to explore overloading the instruction's function bits to store the additional registers (see Figure 6). Using this scheme, it is possible to allocate two additional registers using R3 format and one extra using R4 format. Note that extending the number of operands presents two important drawbacks, 1) it reduces the total number of instructions that can be allocated for the given custom operand. 2) increasing the number of source operands has a dependency on the supported register file in the existing pipeline as reading an additional register adds complexity to the backend.

4.2.3 Control Status Registers. The last option is to split the execution into two parts and pass the extra operands via CSRs before invoking the instruction. The RISC-V specifications enforce no-reordering during the execution of CSR instructions. This solution is the least efficient because 1) CSR's execution stalls the pipeline, and 2) it adds an additional instruction to the pipeline for each invocation of the extension, which will affect the overload IPC.

4.3 Software Support

A the software layer, applications need to be given access to the new instructions. The preferred solution is to implement the new ISA extension into the compiler. However, this requires technical knowledge of the tools and may not be an effective investment if the ISA is subject to changes in the future, a typical reality. The main advantage of supporting the new ISA at the compiler level is to leverage possible high-level code transformations that could automatically be lowered into calls to the extension. The other advantage is the debugging benefits of a supported assembler and disassembler. The alternative and practical alternative to the compiler is to use intrinsic functions that wrap the encoded instructions bytecode. Listing 1 illustrates a simple intrinsic function that wraps the texture sample instruction (Tex) to expose the new instruction to the application.

Listing 1: Tex instruction intrinsic function

```
1  inline tex(u, v, lod) {
2    unsigned __r;
3    asm volatile (".insn␣r4␣0x5b,0,0,%0,%1,%2,%3" :
4       "=r"(__r) : "r"(u), "r"(v), "r"(lod));
5    return __r;
```

5 IMPLEMENTING EXTERNAL EXTENSIONS

Implementing external extensions presents unique challenges in that the hardware is shared by multiple processing cores.

5.1 Local Agents

Local agents are lightweight modules inside the core that manages the communication channel with an external extension. Agents process instructions issued by the pipeline and schedule them for execution on the target external accelerator. Agents also manage per-core local storage if in use. Figure 1 shows the two agents A and B that manage communication with external unit A and B, respectively.

5.2 Arbitration

Arbitrating access to the shared hardware extension can be handled using elastic multiplexer or demultiplexer depending on whether the extension is a consumer or a producer, respectively. The elasticity will enforce proper handshaking as data enter or exit the core. Figure 1 shows the demultiplexer block for the producer unit A feeding the cluster of cores, and the multiplexer select the input for consumer unit B.

6 HARDWARE PERFORMANCE COUNTERS

Hardware performance monitoring counters are essential for profiling and debugging hardware features. There are only 32 of those counters defined within the RISC-V ISA, and three of those counters

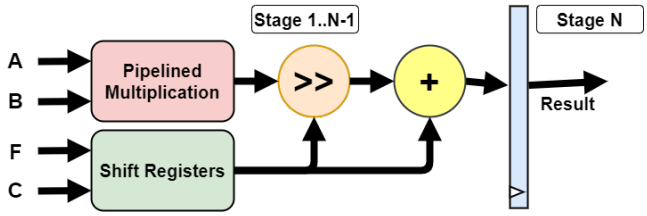

Figure 9: Fixed-point mutliply-add hardware

Figure 10: Fixed-point mutliply-add H/W vs S/W

are reserved. When implementing these counters to profile hardware extensions, it is easy to run out of them. Even without any onboard extension just instrumenting the standard pipeline may often require more counters. We realized that a simple data cache profiling would often use at least half of those counters already.

6.1 Hardware Implementation

A workaround to expand the counters limit is to classify the counters into categories. The preferred classification it to group them based on the component. For instance, the counters for processor pipeline, instruction cache, and data cache will be assigned class 0, 1, and 2, respectively. The implementation can then use one of the non-standard available CSR slots to select the counter class. Figure 8 shows an implementation of that mapping using a simple multiplexer. Note that HPM slots 0-3 are excluded from the multiplexer since they are reserved.

6.2 Software Support

A naive solution to gather hardware performance counters is to dump their content directly to memory. Implementations can reserve an address space where to read the information from the host CPU. It is preferable to flush the counters after the program termination to capture the maximum trace. An implementation may execute the dump in the *_exit()* call after the program *main()* function has returned.

7 SAMPLE IMPLEMENTATION

In this section, we present two non-standard RISC-V extensions we implemented to accelerate Fixed-Point Integer Multiply-Add on the multi-core Vortex GPU [20]. Fixed-point math is commonly used in graphics to approximate floating-point computation. It is often used to in the rasterizer, during interpolation, and even during texture sampling. This justify the need for a fixed-point matrix multiply custom instruction.

7.1 ISA Encoding

Fixed-point mutliplication-add operation is defined as *f(a, b, c, F) = ((a * b) » F) + c* where F is a constant shift parameter. To encode this instruction in RISC-V we use R4 three-source-operands format. We observed that constant F did not vary a lot; in fact, the common values are 8, 16, and 24. Our encoding scheme stores F into the 2-bit *func2* field by encoding its value as follows: 0->0, 1->8, 2->16, 3->24, which is (*func2* « 3). This encoding scheme enabled the same hardware to be used to perform standard integer multiplication, which adds more value to the extension.

7.2 Microarchitecture

Figure 9 shows the basic microarchitecture of the hardware extension. It consists of a pipelined multiplier block that processes A and B operands across several cycles (four cycles in this case). In parallel, operands F, and C are scheduled into a shift register for their output to synchronize with the multiplier where the shift and addition occur.

7.3 Evaluation

Figure 10 compares the software versus the hardware extension runtime execution of the kernel running multiple invocations of imadd instruction on Arria 10 FPGA running at 200 MHz. There is good scaling on both hardware and software as the number of cores increases. Our hardware extension outperforms the software implementation by about 17-20 %, not a huge gain mostly because the source operands are read from memory which introduces a latency overhead inside the kernel.

8 RELATED WORK

Austin et al [1] propose an implementation of the RISC-V Cryptography extension on the Vortex GPU [20]. The authors used the official specifications draft to derive the instructions encoding and implementation. Their hardware addition is located inside each core as part of the execute unit. They achieved a 6.6x and 1.6x speedup in hardware acceleration for AES-256 and SHA-256 respectively compared to the software implementation.

Kuo et al [11] propose a RISC-V extension for accelerating Galois Field arithmetic in cryptography. They implemeted a non-standard extension with four new instructions *FFWIDTH*, *CLMUL*, *CLMULH*, and *FFRED*. Their hardware addition is located inside the execute unit of the processor. They achieved 77% in clock-cycle reduction compared to software implementation.

9 CONCLUSION

In this paper, we discussed the various design challenges of implementing a custom hardware extension in a RISC-V processor. We proposed and contrasted various implementation alternatives based on performance and cost. The RISC-V ISA integrates a rich set of features that enables and encourages the expansion of the platform via its custom instruction support, non-standard CSRs, and hardware performance monitoring counters. It is up to the

architect to make effective use of those facilities understanding the cost-benefits of the design space.

REFERENCES

[1] A. Adams, P. Gupta, B. Tine, and H. Kim, "Cryptography acceleration in a risc-v gpgpu," 2021.

[2] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore, "Efficient 8-bit quantization of transformer neural machine language translation model," *arXiv preprint arXiv:1906.00532*, 2019.

[3] T. Blaise, S. Lee, J. Vetter, and H. Kim, "Cbringing opencl to commodity risc-v cpus," in *Fifth Workshop on Computer Architecture Research with RISC-V*, 2021.

[4] J. Burgess, "Rtx on—the nvidia turing gpu," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020.

[5] M. A. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI," *CoRR*, vol. abs/1906.00478, 2019.

[6] S. Collange, "Simty: generalized simt execution on risc-v," in *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017, p. 6.

[7] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 365–376.

[8] P. Gupta, "Intel xeon+fpga platform for the data center," http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf.

[9] G. Hofemeier and R. Chesebrough, "Introduction to intel aes-ni and intel secure key instructions," *Intel, White Paper*, vol. 62, 2012.

[10] P. Jaaskelainen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "Pocl: Portable computing language," *International Journal of Parallel Programming*, pp. 752–785, 2015.

[11] Y.-M. Kuo, F. Garcia-Herrero, and J. A. Maestro, "Versatile risc-v isa galois field arithmetic extension for cryptography and error-correction codes," 2021.

[12] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, Sep. 2014, pp. 199–202.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[14] C. Lomont, "Introduction to intel advanced vector extensions," Intel White Paper, 2011.

[15] M. Mantor, "Amd radeon™ hd 7970 with graphics core next (gcn) architecture," in *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 2012, pp. 1–35.

[16] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming simd extensions on the pentium iii processor," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, Jul. 2000. [Online]. Available: https://doi.org/10.1109/40.865866

[17] R. Sommefeldt, "A look at the powervr graphics architecture: Tile-based rendering," 2015.

[18] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.35

[19] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[20] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 754–766.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," EECS Department, UC Berkeley, Tech. Rep., 2014.

[22] A. Zeh, A. Glew, B. Spinney, B. Marshall, D. Page, D. Atkins, K. Dockser, M.-J. O. Saarinen, N. Menhorn, R. Newell, and C. Wolf, "RISC-V cryptographic extension proposals volume i: Scalar & entropy source instructions," https://github.com/riscv/riscv-crypto/releases/tag/v0.9.0-scalar, Mar. 2021.