Bringing OpenCL to Commodity RISC-V CPUs

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

Seyong Lee
Oak-Ridge National Laboratory
Oak-Ridge, Tennessee
lees2@ornl.gov

Jeff Vetter
Oak-Ridge National Laboratory
Oak-Ridge, Tennessee
vetter@ornl.gov

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

ABSTRACT

The importance of open-source hardware has been increasing in recent years with the introduction of the RISC-V Open ISA. This has also accelerated the push for support of the open-source software stack from compiler tools to full-blown operating systems. Parallel computing with today's Application Programming Interfaces such as OpenCL has proven to be effective at leveraging the parallelism in commodity multi-core processors and programmable parallel accelerators. However, to the best of our knowledge, there is currently no publicly available implementation of OpenCL targeting commodity RISC-V processors that is accessible to the open-source community. Besides opening RISC-V to the existing rich variety of scientific parallel applications, OpenCL also provides access to a unique genre of benchmarks useful in computer architecture research. In this work, we extended an Open-source implementation of OpenCL to target RISC-V CPUs. Our work not only cover commodity multi-core RISC-V processors, but also plethora of low-profile embedded RISC-V CPUs that often do not support atomic instructions or multi-threading.

KEYWORDS

High-Performance Computing, multi-threading, heterogeneous Computing, Parallel Programming, Compiler Optimizations.

ACM Reference Format:
Tine Blaise, Seyong Lee, Jeff Vetter, and Hyesoon Kim. 2021. Bringing OpenCL to Commodity RISC-V CPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The current challenges in technology scaling [9] are pushing the semiconductor industry towards hardware specialization, creating a proliferation of heterogeneous systems-on-chip, delivering orders of magnitude performance and power benefits compared to traditional general-purpose architectures. This transition is getting a significant boost with the advent of RISC-V [24] with its unique modular and extensible ISA, allowing a wide range of low-cost processor designs for various target applications. As these processors become specialized, often extending the existing ISA with domain-specific instructions, a challenge arises on how to program them and take advantage of their new capabilities. OpenCL [17] is currently the most widely adopted programming framework for heterogeneous platforms available on mainstream CPUs, GPUs, as well as FPGAs [22], and custom DSPs [11]. Adopting OpenCL across all devices in heterogeneous platforms greatly simplifies the design and implementation of the software stack that drives these platforms, allowing OpenCL applications to scale efficiently as new computing devices are introduced. Additionally, there is a large ecosystem of existing parallel applications written in OpenCL that will now be able to run on those devices.

There is currently no publicly available implementation of OpenCL for RISC-V accessible to the open-source community. This is partly because the vector ISA extension [3] for RISC-V is still under development. However, there are existing implementations of RISCV processors with multi-core support [2], including some with custom ISA for parallel processing [5] [10], [8] that could greatly benefit from OpenCL to increase adoption. Implementing a flexible compiler framework that makes it easy to port OpenCL to RISC-V-based heterogeneous parallel processors is a challenge and this project aims at solving this application programming gap.

There exists a wide range of open-source implementations of OpenCL targeting various architectures, including X86 with Beignet [13], Intel Neo [14], Nvidia with libclc [18], AMD with Clover [23], ROCm [7], TI DSPs with TI-OpenCL [12], ARM with Shamrock [19],

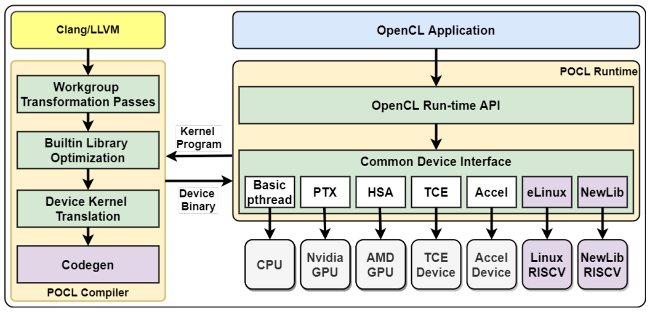

Figure 1: POCL RISC-V System Architecture

and POCL [15]. POCL implements a flexible compilation back-end based on LLVM, allowing it to support a wider range of device targets including X86, ARM, AMD, Nvidia, and TCE [16], making it our preferred choice for implementing a RISC-V back-end. Supporting OpenCL on RISC-V provides several research opportunities. First, OpenCL enables the evaluation of custom parallel processor design leveraging the existing large ecosystem of parallel applications and benchmarks written in OpenCL. Second, it enables the exploration of the design space of our processor including introducing new ISA extensions to achieve desired power and energy targets. Third, it allows the exploration of new application-specific compiler optimization techniques that RISC-V-based parallel processors could leverage to improve performance.

In this work, we extended the POCL [15] Open-source implementation of OpenCL to target RISC-V CPUs. Our work not only cover commodity multi-core RISC-V processors, but also plethora of low-profile embedded RISC-V CPUs that often do not support atomic instructions or multi-threading. An overview of the proposed compilation flow is shown in figure 1, highlighting the POCL compiler and runtime components and our specialization to support RISC-V CPU targets. The purple components are modifications to the baseline system.

This paper makes the following key contributions:

- We detail the implementation of the OpenCL compiler extension for supporting RISC-V linux-capable processors, which are CPU implementations with support for multi-threading and atomic instructions.
- We detail the implementation of the OpenCL compiler extension for supporting RISC-V newlib-capable processors, which are low-profile CPU implementations without atomic instructions or multi-threading support.
- we introduce a new kernel execute technique called kernel static registration which enables the automatic registration of built-in static kernels for OpenCL. A technique that makes it possible to execute multiple OpenCL kernels on low profile devices lacking file system support.
- We present an evaluation of our compiler extension using a series of OpenCL kernels running on the virtualization platform with Fedora operating system.

2 BACKGROUND ON POCL

POCL is an open-source Performance Portable OpenCL Implementation that supports various types of target devices, including general-purpose processors (e.g. x86, ARM, Mips), general-purpose GPU (e.g. Nvidia), and custom TCE-based accelerators [16]. The custom accelerator support provides an efficient solution for enabling OpenCL applications to use hardware devices with specialized acceleration (e.g. SPMV, GEMM). Figure 1 illustrates the system architecture of the POCL framework, which includes a runtime shared library implementing the OpenCL API and an LLVM-based back-end compiler for compiling the kernels.

The POCL runtime implements three distinct execution layers: 1) the actual OpenCL runtime API calls, 2) a device-independent common interface where each device target implementation plug into to specialize particular operations. This device independent layer implements the common operations needed by the devices such as the compiler driver, the file system abstraction, the memory abstraction, threading, and synchronization libraries. 3) a device-specific layer where target hardware dedicated specializations are implemented, such as native compilers , hardware resource management, and kernel execution. There are five classes of target devices currently supported in POCL (Figure 1): 1) The basic or pthread devices which are tailored towards standard multi-core processors such Intel X86 or ARM. 2) PTX device class for Nvidia GPU accelerators. 3) HSA device class for AMD GPU accelerators. 4) TCE device class for compiler-generated application-specific processors. 5) Accel device for fixed-function custom accelerators. When executing on GPU accelerators, POCL compiler will export PTX for NVidia devices and HSAIL [20] IR for HSA compatible devices. At runtime, POCL invokes the back-end compiler of the given GPU class with the provided PTX or HSAIL IR.

The POCL back-end compiler (Figure 1) processes the OpenCL application kernel and applies a custom set of workgroup transformation passes on the kernel to handle barriers and implement low-level device-specific parallelization. There is also a built-in library that implements optimized device-specific OpenCL functions such as transcendental math operations (e.g. sin(), cos(), tan()). The reason for supporting this custom library is to provide more efficient implementations of those routines than what LLVM would produce by default. It also enables direct access to custom device instructions when targeting TCE-based accelerators. POCL supports target-specific execution models, including SIMT, MIMD, SIMD, and VLIW. On platforms supporting MIMD and SIMD execution models such as CPUs, the POCL compiler attempts to pack as many OpenCL work-items in the work-goup to the same vector instruction if SIMD is supported, then the POCL runtime will distribute the remaining work-items among the active hardware threads on the device with provided synchronization using the operating system's threading library. On platforms supporting SIMT execution model such as GPUs, the POCL compiler delegates the distribution of the work-items to the hardware to spread the execution among its various SIMT cores, relying on the device to also handle the necessary synchronization. On platforms supporting VLIW execution models such as TCE-based accelerators, the POCL compiler attempts to "unroll" the parallel regions in the kernel code such that the operations of several independent work-items can be statically scheduled to the multiple function units of the target device.

On CPU device targets, the generated kernel implements workgroup callback functions for each functions in the source kernel program. A workgroup in OpenCL is a collection of workitems to be scheduled for execution on the device, they represent a three dimensional matrix and there are multiple of those workgroups forming another multi-dimensional matrix called NDRange (see Figure 2). Listing 1 illustrates the signature of a kernel call function. This callback function is invoked at runtime by a thread executing on the target device with the kernel arguments, a context object, and the active (x, y, z) workgroup offsets. The kernel context object passed to the call to provide workgroup dimensions. Internally the workgroup callback function implements the workgroup's iteration to invoking each workitem call.

At the end of the compilation phase, POCL will package the compiled kernel into the POCL Binary format. The structure of this binary file is illustrated in Figure 4-A. The structure consists of a

Name	Version	Target	status	Url
Intel Beignet	2.0	x86	Closed	https://cgit.freedesktop.org/beignet
Intel Neo	2.1	X86	Active	https://01.org/compute-runtime
POCL	1.2	x86, ARM AMD, TCE, PTX	Active	https://github.com/pocl/pocl
ROCm	1.2	AMD	Stable	https://github.com/RadeonOpenCompute
TI-OpenCL	1.1	TI	Closed	https://git.ti.com/opencl/ti-opencl
Shamrock	1.2	ARM	Closed	https://git.linaro.org/gpgpu/shamrock.git
libclc	1.2	AMD, PTX	Closed	https://libclc.llvm.org/
Clover	1.1	AMD	Closed	https://people.freedesktop.org/ steckdenis/clover/

Table 1: Open-Source OpenCL Implementations

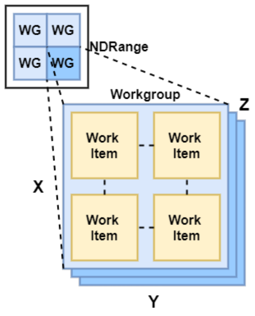

Figure 2: OpenCl Execution Model

device build hash to identify the kernel's identity in association with its target device architecture and compiled device-independent compiled LLVM bytecode of the kernel. This bytecode is extracted at runtime and compiled for the target device using its native compiler.

Listing 1: POCL Kernel Invocation

```
// kernel runtime context
struct pocl_context {
  uint num_groups[3];         // internal use
  uint global_offset[3];      // internal use
  uint work_dim;              // internal use
  uint local_size[3]          // workgroup size
  char* printf_buffer;        // for debugging
};

void run_workgroup(const void* arguments,
                   const pocl_context* context,
                   int x,
                   int y,
                   int z)
{
  // per-workgroup iterations
  for (int i: x -> context->local_size[0]) {
    for (int j: y -> context->local_size[1] ) {
      for (int k: z -> context->local_size[2] ) {
        // workitem (i,j,k) invocation
      }
    }
  }
}
```

3 RELATED WORK

Table 1 summarizes the current status of open-source OpenCL run-time implementations.

Intel Beignet [13] was one of the original first open-source implementations of OpenCL on general-purpose CPUs. The software was only supported on Intel X86 Architecture, taking advantage of its advanced SIMD instruction set to optimize the kernel inner loop. The software is not in active development anymore, being replaced by the current Intel Neo.

Intel Neo [14] is the latest implementation of an open-source parallel API by Intel that supports both OpenCL and Level-Zero. Level Zero is Intel's bare-metal software interfaces that provide application direct access to the processor's acceleration services for fine-grain performance optimizations. Similar to Beignet, Intel Neo only targets Intel architecture.

Clover [23] and ROCm [7] are open-source implementation of OpenCL targeting AMD GPUs. Clover is limited to OpenCL version 1.1 and is not anymore in active development. ROCm is actively maintained by AMD but simular to Intel Neo, the software support is restricted to AMD processor architectures.

Shamrock [19] is an open-source implementation of OpenCL targeting ARM. It is an adapted fork of the Clover framework with no further development beyond OpenCL 1.2 support.

TI-OpenCL [12] is an open-source implementation of OpenCL targeting Texas Instruments's Digital Signal Processors. The device target for this implementation is too restrictive for a possible consideration to extend the support for RISC-V.

Libclc [18] is an open-source implementation of OpenCL targeting Nvidia GPUs. The framework compiles OpenCL into PTX IR for export to Nvidia native compiler. Its target device base is restricted to PTX, and future development has halted.

POCL presents the most promising path for extending a support for RISC-V since it supports the largest number of instruction set architectures, showing its flexibility in adapting for a new architecture. Also, POCL implements the support for ARM, which uses cross-compilation similar to RISC-V. Another advantage that POCL has is its LLVM-based back-end compiler which makes it easier to implement custom optimization passes. Lastly, POCL is still in active development with new features being added to support OpenCL capabilities beyond version 1.2.

4 OPENCL SUPPORT FOR LINUX-CAPABLE RISC-V CPUS

The Linux-capable RISC-V CPUs support the necessary ISA capabilities to run the Linux operating system, specifically atomic, control status registers, and fence instructions, described via extensions A, Zicsr, and Zifencei, respectively. The following modifications to the POCL framework were necessary to support Linux-capable RISC-V CPUs: 1) Registering a new device target for the RISC-V Linux class such that users for the framework will use the configuration class for their specific hardware. This new target is a derivation of the existing CPU device class that uses the operating system threading model to execute the kernel. 2) Enabling the runtime to compile into a RISC-V shared library via cross-compilation 3) Adding a new mode of execution for offline kernel compilation. Figure 3 illustrates the new compilation pipeline for Linux-capable RISC-V CPUs on

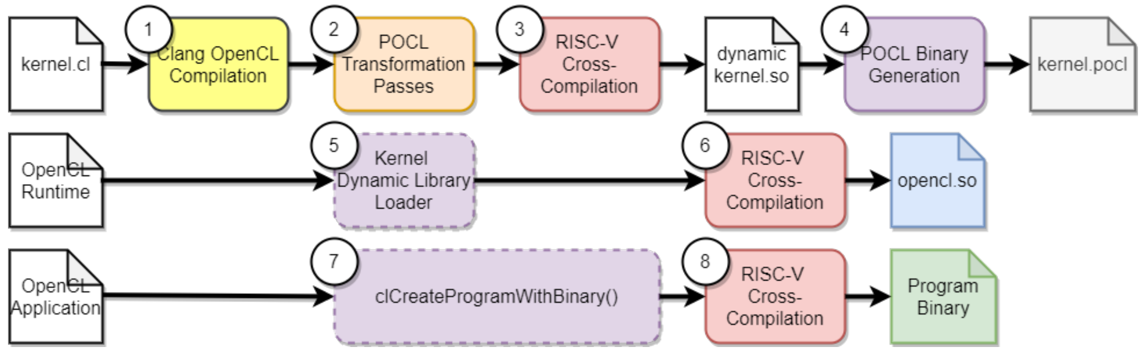

Figure 3: Compilation Pipeline for Linux-Based RISC-V Support

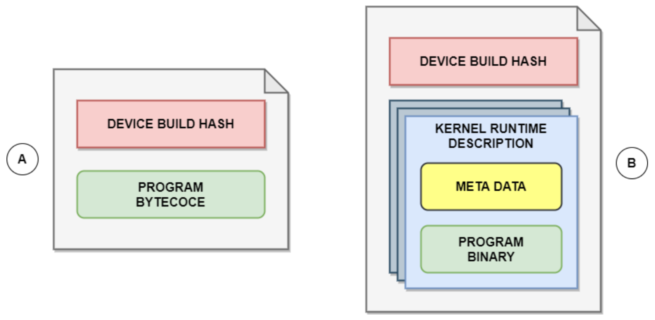

Figure 4: (a) Original POCL Binary Structure. (b) RISC-V POCL Binary Structure.

POCL showing three separate flows for the kernel, the OpenCL runtime, and the OpenCL application. The Kernel compilation follows four stages: 1) the Clang compilation where the OpenCL kernel to converted to LLVM IR. 2) the POCL transformation passes where built-in functions optimizations and barriers handling take place. 3) cross-compilation of the transformed kernel to RISC-V shared library. 4) Packaging of the kernel into POCL binary format to be loaded at runtime. For the OpenCL runtime compilation flow, we modified the kernel launcher to load the pre-compiled shared library directly from the POCL binary file (5) instead of compiling it at run time as default. On the application side, *clCreateProgramWithBinary()* is now used instead of the default *clCreateProgram()* API to directly load the kernel from the POCL binary file (7).

4.1 Offline kernel Compilation

The most important change needed in POCL to support RISC-V Linux CPU was the support for offline compilation, the ability to pre-compile the kernel on the host computer and load it at runtime during the application execution. The current implementation of POCL compiles the kernel to native binary at runtime, and only the OpenCL to LLVM transformation can be operated offline. This design was done mainly to support portability with multiple platforms. To support offline native compilation, we modified the POCL binary file format to store pre-compiled kernel binaries (Figure 4-B) instead of the original LLVM program bytecode (Figure 4-A). The new file format keeps the device hash and an array of kernel descriptions with metadata and the pre-compiled shared library. We need an array because a single OpenCL kernel file may contain multiple kernel functions which are compiled separately. The kernel metadata consist of the information needed to build the POCL runtime context used when invoking the kernel (listing 1)

5 OPENCL SUPPORT FOR NEWLIB-CAPABLE RISC-V CPUS

The Newlib-capable RISC-V CPUs do not support the necessary ISA capabilities to run the Linux operating. These low-profile embedded RISC-V processors are ubiquitous in the domain of IoT and edge computing. This class of CPUs uses the Newlib [6] library as a system platform for building and running applications on the device. Newlib provides a lightweight implementation of the C standard library in which requirements for the file system and other I/O services have been removed. Supporting Newlib-capable CPUs on POCL presents several challenges: 1) The lack of a file system means that the application will have to be packaged and distributed differently. 2) The lack of multi-threading capabilities means that kernel execution will have to take place on the application thread. To address the lack of a file system support on NewLib-capable CPUs, we devised a new technique called static kernel registration to enable the OpenCL runtime to run on Newlib-capable CPUs and support applications with multiple kernels. The following modifications to the POCL framework were necessary to support Newlib-capable RISC-V CPUs: 1) Registering a new device target for the RISC-V Newlib class such that users for the framework will use the configuration class for their specific hardware. 2) Adding a codegen pass to the back-end compiler to support static kernel registration. 3) Modifying the OpenCL runtime to support static kernel registration.

Figure 5 illustrates the new compilation pipeline for Newlib-capable RISC-V CPUs on POCL, showing three separate flows for the kernel, the OpenCL runtime and the OpenCL application. In the kernel compilation flow, we added a new stage (3) after POCL transformation passes where we transform the kernel call interface to support automatic static registration. For the OpenCL runtime compilation flow, we modified the kernel loading module to support kernel lookup (5) and invocation (7). The OpenCL applications

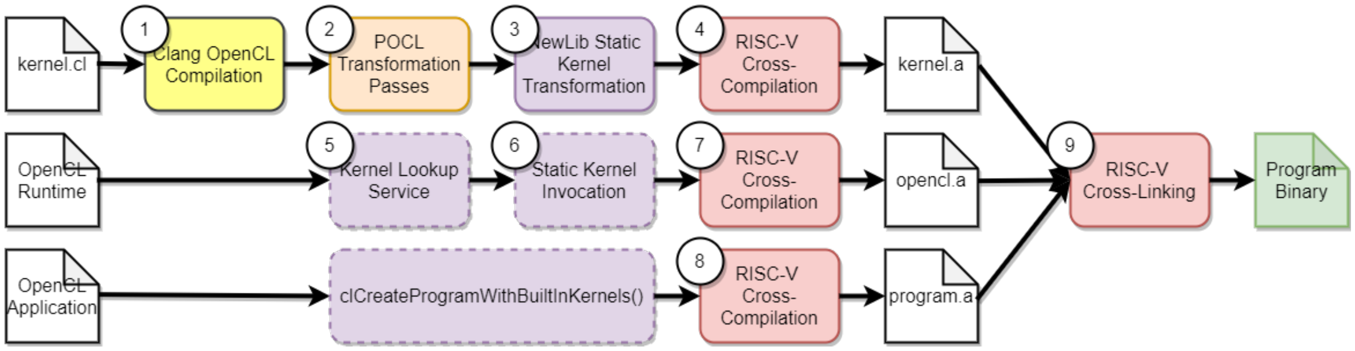
kernel.cl
1 Clang OpenCL Compilation
2 POCL Transformation Passes
3 NewLib Static Kernel Transformation
4 RISC-V Cross-Compilation
kernel.a
OpenCL Runtime
5 Kernel Lookup Service
6 Static Kernel Invocation
7 RISC-V Cross-Compilation
opencl.a
9 RISC-V Cross-Linking
Program Binary
OpenCL Application
clCreateProgramWithBuiltInKernels()
8 RISC-V Cross-Compilation
program.a

Figure 5: Compilation Pipeline for Newlib-capable RISC-V Support

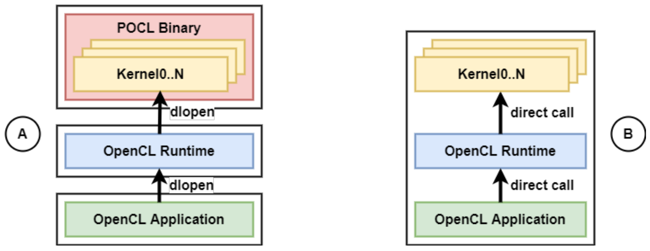

Figure 6: Kernel Invocation Sequence Comparison between Linux (A) vs Newlib (b) POCL Runtime Implementations

running on NewLib RISC-V now use the **clCreateProgramWithBuiltinKernels()** OpenCL API to load any registered static kernel at run time. Because Newlib-capable CPU devices cannot host a file system, we statically link the kernel together with the OpenCL runtime and the application into a single program binary (9) that is executed on the device as a standalone executable. Figure 6 contrasts the difference between the Linux vs NewLib kernel invocation sequences, showing how all components in NewLib are contained in a single executable making internal direct calls between them.

5.1 Static Kernel Registration

Static kernel registration addresses the challenge of static kernel invocation inside POCL with multiple functions. In POCL, each kernel function is compiled separately, and a unique workgroup runtime function is generated for each. Listing 2 shows simple OpenCL kernel with multiple function entries *Foo* and *Bar*, the POCL compiler will generate separate workgroup runtime functions *run_workgroup_Foo()* and *run_workgroup_Bar()*, respectively.

Listing 2: OpenCL kernel with multiple functions

```
__kernel void Foo(__global float *dst,
                  __global const float *src,
                  int size)
{...}

__kernel void Bar(__global float *dst,
                  __global const float *src,
                  int size)
{...}
```

Our new compiler transformation inserts a kernel registration block into the kernel static library to instantiate the workgroup functions registration. Listing 3 illustrates the speudo-code of the registration block. The main concept behind this technique is to exploit C++'s automatic construction of static objects [1]. Using an unnamed namespace, we define a local class for each kernel function with a default constructor that calls into the external *__register_static_kernel()* routine to register itself, providing the arguments and local variables needed to call the workgroup function.

Listing 3: Static kernel auto-registration

```
int __register_static_kernel(...);
void run_workgroup_Foo(...);

namespace {
  class auto_register_kernel_t {
  public:
    auto_register_kernel_t() {
      static int arg_types[] = { T0, T1, ... };
      static int local_sizes[] = { S0, S1, ... };
      __register_static_kernel(
        "Foo",
        run_workgroup_Foo,
        <arguments>,
        <locals>);
    }
  };
  static auto_register_kernel_t __Foo__;
}
```

5.2 Running Statically Compiled OpenCL Applications

We also modified the OpenCL runtime in POCL such that we can support static registration of kernel functions. Listing 4 shows the speudo-code of the static kernel support routines that were added to the OpenCL runtime. Starting with the *__register_static_kernel()* routine implementation, referenced earlier during the kernel compilation (Listing 3), this code allocates new static kernel entries for each registered functions into a global static table. The next routine *__query_static_kernel()* handles the query of the registered kernels by iterating thru each item in the global table to match the input name. The query mechanism is integrated with the public *clCreateProgramWithBuiltinKernels()* OpenCL API call using POCL's internal API Calls *pocl_newlib_supports_builtin_kernel()* and *pocl_newlib_get_builtin_kernel_metadata()*.

At rumtime, when the OpenCL application calls *clCreateProgramWithBuiltinKernels()*, POCL will first call *pocl_newlib_supports_builtin_kernel()* to see if the specified kernel function is currently registered, then will call *pocl_newlib_get_builtin_kernel_metadata()* to load the kernel metadata necessary for the workgroup function invocation (callback function, arguments, and local variables). Later, when the application is ready to execute the kernel, the kernel's workgroup callback function is invoked.

Listing 4: Static kernel Runtime Registration

```
struct kernel_info_t;

static int g_num_kernels = 0;
static kernel_info_t g_kernels [MAX_KERNELS];

void __register_static_kernel(...) {
  auto kernel = g_kernels + g_num_kernels++;
  kernel->name = name;
  kernel->callback = callback;
  kernel->arguments = arguments;
  kernel->locals = locals;
}

bool __query_static_kernel(...) {
  for (auto kernel : g_kernels) {
    if (strcmp(kernel->name, name) != 0)
      continue;
    *callback = kernel->callback;
    *arguments = kernel->arguments;
    *locals = kernel->locals;
    return true;
  }
  return false;
}

bool pocl_newlib_supports_builtin_kernel(...) {
  return __query_static_kernel(kernel_name, ...);
}

void pocl_newlib_get_builtin_kernel_metadata(...) {
  __query_static_kernel(kernel_name, ...);
}
```

6 EVALUATION

6.1 Experimental Setup

To evaluate our implementation, we use QEMU [4] virtualization platform targetting RISC-V 64-bit. The QEMU system memory configuration was 2 GB. We used Federa operating system version 32. The guest operating system was Ubuntu 18.04 running a 4-core Intel i7 with 16GB of system memory. We have classified the benchmarks into a compute bounded group that includes *sgemm*, *vecadd*, *sfilter*, and *sxapy*, and a memory bounded group that includes *nearn*, *gaussian*, and *bfs*. Table 2 shows the RISC-V CPU configurations that we tested. The lowest profile with NewLib support and no floating-point hardware was able to execute on the Spike [21] simulator and QEMU. This configuration is well suited for embedded devices in the IoT sector.

Figure 7 shows our OpenCL benchmark suite executing on QEMU for three configurations, the Newlib-RISCV POCL, the Linux-RISCV POCL on 4 threads, and the Linux-RISCV POCL on 8 threads, the runtime latency is in milliseconds. All applications running on Newlib were compiled statically as standalone binaries with their kernel and the POCL runtime. The Linux-capable applications load the POCL runtime and kernel from shared libraries. The Newlib applications are much slower as expected because it is single-threaded and assumes an architecture without multi-core support. Increasing the number of threads in QEMU improves the execution time for the Linx-RISCV POCL across all applications. We use QEMU mainly for validation that the system works. However, on actual physical hardware, these OpenCL applications' performance will depend on the underlying microarchitecture it will be running on.

RISC-V Architecture	POCL Device Class	Tested Environments
RV32im -lp32	RISCV-NewLib	Spike, QEMU
RV32imf -lp32f	RISCV-NewLib	Spike, QEMU
RV64imfd -lp64d	RISCV-NewLib	QEMU
RV32gc -lp32d	RISCV-Linux	QEMU
RV64gc -lp64d	RISCV-Linux	QEMU

Table 2: Tested RISC-V CPUs Configurations

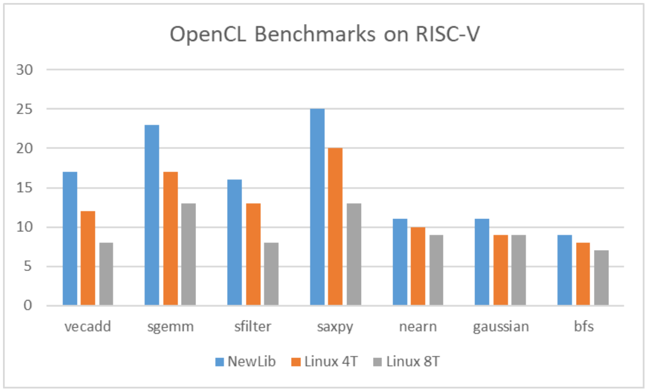

Figure 7: OpenCL Benchmarks Runtime Latency (ms) on RISC-V 64-bit QEMU

7 CONCLUSION

Expanding the RISC-V ecosystem to include parallel applications via OpenCL opens the door to the large segment of scientific computing. Furthermore, it provides hardware designers new workloads for exploring new microarchitecture designs and optimizations. In this work, we proposed an implementation of this capability by extending the POCL framework to support RISC-V. Our extension particularly targets today's large segment of commodity RISC-V cores without SIMD vector extensions and also expands to low-profile embedded processors with minimal capabilities. We validated our design using Spike and QEMU virtualization platform. This work is being made available as an open-source project for public use.

REFERENCES

[1] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, 2001.
[2] AndesCore, "Andescore ax25mp multicore." [Online]. Available: http://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax25mp
[3] K. Asanovic, *RISC-V Vector Extension.* [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

[4] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[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] J. J. Corinna Vinschen, "Newlib," http://sourceware.org/newlib, 2001.

[7] A. M. Devices, "Rocm platform," https://rocm.github.io.

[8] F. Elsabbagh, B. Asgari, H. Kim, and S. Yalamanchili, "Vortex risc-v gpgpu system: Extending the isa, synthesizing the microarchitecture, and modeling the software stack," https://carrv.github.io/.

[9] 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.

[10] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 670–684.

[11] T. Instruments, "Digital signal processors (dsp) opencl." [Online]. Available: http://www.ti.com/processors/digital-signal-processors/libraries/OpenCL.html

[12] ——, "Opencl for texas instruments dsps." [Online]. Available: https://git.ti.com/cgit/opencl/ti-opencl

[13] Intel, "Beignet opencl library," https://cgit.freedesktop.org/beignet.

[14] ——, "Intel compute runtime," https://01.org/compute-runtime.

[15] 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.

[16] P. O. Jäskeläinen, C. S. de La Lama, P. Huerta, and J. H. Takala, "Opencl-based design methodology for application-specific processors," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2010, pp. 223–230.

[17] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.1*, 2011. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[18] M. Larabel, "Libclc: An opencl c library implementation," https://libclc.llvm.org.

[19] G. Pitney, "Shamrock: An opencl implementation based on clover," https://git.linaro.org/gpgpu/shamrock.git.

[20] B. Sander and A. S. FELLOW, "Hsail: Portable compiler ir for hsa." in *Hot Chips Symposium*, 2013, pp. 1–32.

[21] SiFive, "Spike risc-v isa simulator," https://github.com/riscv/riscv-isa-sim, 2018.

[22] D. Singh, "Implementing fpga design with the opencl standard," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01173-opencl.pdf, 2011.

[23] D. Steckelmacher, "Clover: Opencl 1.1 software implementation," https://people.freedesktop.org/~steckdenis/clover.

[24] 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.