Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC

Anthony Cabrera, Seth Hitefield, Jungwon Kim, Seyong Lee, Narasinga Rao Miniskar, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6173
Email: {cabreraam, hitefieldsd, kimj, lees2, miniskarnr, and vetter}@ornl.gov

***Abstract*—Future heterogeneous domain-specific systems on a chip (DSSoCs) will be extraordinarily complex in terms of processors, memory hierarchies, and interconnection networks. To manage this complexity, architects, system software designers, and application developers need programming technologies that are flexible, accurate, efficient, and productive. These technologies must be as independent of any one specific architecture as is practical because the sheer dimensionality and scale of the complexity will not allow porting and optimizing applications for each given DSSoC. To address these issues, the authors are developing Cosmic Castle, a performance portable programming toolchain for streaming applications on heterogeneous architectures. The primary focus of Cosmic Castle is on enabling efficient and performant code generation through the smart compiler and intelligent runtime system. This paper presents the preliminary evaluation of the authors' ongoing work toward Cosmic Castle. Specifically, this paper details the code-porting efforts and evaluates various benchmarks on the Qualcomm Snapdragon SoC using tools developed through Cosmic Castle.**

I. Introduction

Heterogeneous and manycore processors are becoming the de facto architectures for both traditional high-performance computing (HPC) systems and domain-specific systems on a chip (DSSoCs). One critical challenge of these architectures is programmability. Numerous active efforts are working to improve programmability and portability, including directive-based programming models [1], [2], library application programming interfaces (APIs) [3], [4], and parallel programming languages [5]. However, experiences with GPUs [6], [7] and Knights Landing [8] illustrate how difficult it is to obtain performance close to that offered by the native programming environment for a given system. Rapidly changing system architectures also further complicate this issue.

Several programming solutions of varying maturity and performance have been developed to, in part, enable performance portability. Notable solutions include OpenCL [5], [9], OpenACC [1], [6], [10], [11], and OpenMP [2], [12]–[14]. Library and meta-programming solutions (e.g., cuBLAS [15], Thrust [16], Legion [3], Kokkos [4]) have also demonstrated promise. Still, programming these systems may require significant architectural expertise, and the current solutions often lack performance portability. Domain-specific languages are another approach for addressing productivity, performance, and portability issues in programming modern architectures [17]–[21]. However, these approaches require that all of a given application's kernels must be rewritten with domain-specific languages and often focus on specific architectures that lack mature programming environments or community support.

To this end, the authors present Cosmic Castle, a software ecosystem that provides performance portable programming solutions for heterogeneous and manycore processors and deep memory hierarchies while balancing against programmability. To achieve this goal, the authors' proposed solution leverages existing efforts as much as possible by integrating software tools from vendors and the community into the ecosystem to find the best portable, performant, and sustainable solutions. This is an ongoing project. This paper focuses on the main programming system of Cosmic Castle and evaluates its effectiveness and potential as a performance portable programming system. The Qualcomm Snapdragon line of SoCs is used as a case study. This paper details the process of integrating the existing Snapdragon tooling with the authors' tools in Cosmic Castle, and porting several benchmarks from various application domains onto the Snapdragon SoC.

The main contributions of this paper are as follows.

- This paper describes Cosmic Castle, a performance portable programming toolchain for porting applications on heterogeneous architectures, which consists of the smart compiler, Open Accelerator Research Compiler (OpenARC) [10], and the Intelligent Run Time System (IRIS) [22].
- This paper provides an example of using Cosmic Castle on the Qualcomm Snapdragon SoC in terms of its architectures and their implications on programming with emphasis on the Hexagon digital signal processor (DSP).
- This paper evaluates the effectiveness and potential of the proposed Cosmic Castle programming system by porting several benchmarks from different application domains onto the Snapdragon SoC.

II. Qualcomm Snapdragon Architecture and Programming

The Qualcomm Snapdragon line of SoCs is a versatile embedded processor that includes powerful ARM CPU cores along with accelerators such as GPU and DSPs. It has been a viable solution for developing mobile applications, such as multimedia/vision processing, 5G networking, artificial intelligence, and Internet Protocol cameras. The Snapdragon 855 and 865 chipsets are widely used in commercial mobile platforms (e.g., smartphones and tablets) by vendors such as Samsung, LG, Motorola, OPPO, and Xiomi.

A. Architecture and Programming

Qualcomm Snapdragon 855 and 865 provide heterogeneous compute capability at different levels, as shown in Figure 1. They offer different variants of compute units, such as Dynamic IQ `big.LITTLE` multicore ARM cores, Adreno GPU, Hexagon DSP, and Spectra 360 (480 for 865 chipset). These compute elements are targeted through different programming models, such as C/C++ with OpenMP for multicore ARM cores; OpenCL, OpenGL, Vulkan, and DirectX for GPU; and C/C++ and assembly language for Hexagon DSP.

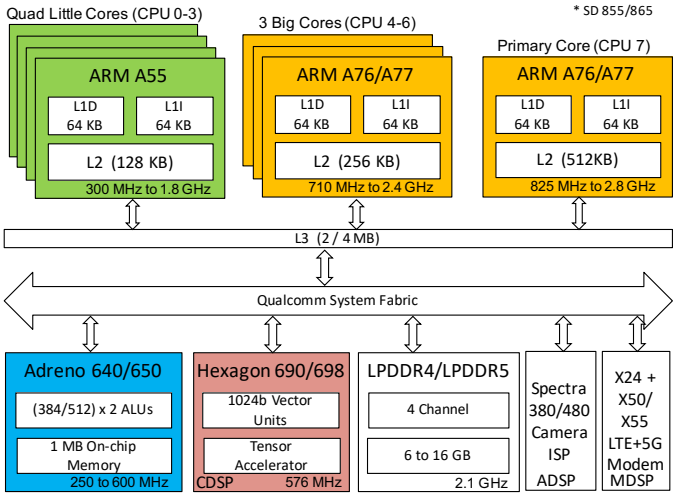

Fig. 1: Snapdragon 855 and 865 with heterogeneous compute units.

1) `big.LITTLE` ARM CPUs

The Snapdragon 855 and 865 chipsets include the Kryo series of CPUs, which are based on ARM-based CPUs. Although there are eight ARM-based cores, they are not symmetric. The Snapdragon chipset has four low-power Cortex-A55 cores (i.e., `LITTLE` cores) and four high-performance Cortex-A77/A76 cores (i.e., `big` cores), and one of these cores acts as the "prime" core. Moreover, the cache sizes of these cores are also different. The prime core has a dedicated 512 KB L2 cache, whereas each of the other `big` cores have a dedicated 256 KB L2 cache. Multithreading of these cores can be exploited through the OpenMP programming model or by using the `pthread` library.

The `LITTLE` cores (cores 0–3) are operable at 18 frequency scale modes ranging from 300 MHz to 1.8 GHz. The `big` cores (cores 4–6) can operate at 17 frequency scale modes ranging from 710 MHz to 2.4 GHz. The prime core (core 7) can run at 20 frequency scale modes ranging from 825 MHz to 2.8 GHz. The frequency of each core can be set at run time or through the Linux shell.

2) Adreno GPU

The Snapdragon 855 chipset also contains the Adreno 640 GPU, which is used for mobile chipsets and was developed by Qualcomm. It is used for 2D/3D graphics acceleration. The Adreno 640 GPU has 384 $\times$ 2 arithmetic logic units (ALUs) and supports unified memory addressing. It has 1 MB on-chip graphic memory, and its frequency can be scaled from 250 to 600 MHz. However, it is unclear how its frequency can be changed through programming.

3) Hexagon DSP, NPU, and Spectra ISP

The Spectra 360 is an image and signal processor (ISP), which is a hardware accelerator for computer vision. Hexagon DSP is used to efficiently process multimedia algorithms. It also has a tensor accelerator and a dedicated neural processing unit (NPU) for efficiently processing deep neural networks. The core Hexagon DSP is a Very Large Instruction Word (VLIW) architecture and is supported with hardware multithreading. There are three variants of Hexagon DSPs available: the audio DSP (ADSP) for audio processing, compute DSP (CDSP) for compute-intensive tasks, and modem DSP (MDSP) for X24 LTE/5G modems. The CDSP has 1,024 bit vector functional units. The ADSP and CDSP are programmable through C/C++ and assembly language. CDSP vector functional units require special Hexagon Vector Extensions (HVX) instructions. These instructions are either manually embedded in the hexagon kernel (C/C++) or can be generated by the auto-vectorizing compiler feature provided by the Qualcomm Hexagon software development kit (SDK) toolchain. Spectra ISP, MDSP, and NPU can be used through dedicated APIs provided by Qualcomm.

§II-B discusses optimization of the Snapdragon Hexagon DSP in more detail.

B. Designing and Optimizing Applications for Hexagon DSP

As described in §II-A3, the Hexagon DSPs in the Snapdragon SOC are VLIW processors equipped with a 1,024 bit vector coprocessor. The benefit of offloading computation to Hexagon DSPs is that the hardware is significantly less complex than the ARM CPUs and are clocked slower. Paired with the potential for vectorized computation, this enables the possibility of accelerated computation relative to the CPUs while also consuming less power. This is particularly beneficial for enabling heavier computation workloads at the edge.

However, the Hexagon DSPs are notoriously difficult to program. Although the Snapdragon CPU and GPU can be targeted by using OpenMP and OpenCL, respectively, no such model exists for the DSP. Additionally, the DSP is a VLIW processor, and writing efficient code to target this processing paradigm requires architecture-specific knowledge of the DSP. Finally, the documentation and example programs for the Hexagon DSP are not as rich as other vectorized

compute solutions, such as the Intel Streaming SIMD (single instruction, multiple data) Extensions and Advanced Vector Extensions or the ARM Neon vector extensions. For example, Intel and ARM both have online, user-friendly documentation of their vector intrinsics for their various vector units [23], [24]. Although examples exist in the Hexagon SDK, they are not well documented, and choices made in the source code are not immediately obvious. Additionally, the Hexagon DSP SDK is a framework entirely separate from the native Android build environment, which is required to build applications that target the platform. This additional framework is yet another thing that must be learned to build applications for this platform.

This section decreases the sparsity in the literature regarding how to program the Hexagon DSP and shows the authors' progress toward designing and optimizing applications that leverage the Hexagon DSP.

1) Creating a Hexagon DSP Application

FastRPC: Fast remote procedure call (FastRPC) is the framework that allows transparent remote calls from the host processor to the DSP. Specifically, this is what enables users to offload computation to the DSP. This is done by generating a `stub` and skeleton, or `skel`, library that handles communication between the host and device. The `stub` and `skel` libraries reside on the host and DSP sides, respectively, and are autogenerated based on the user-provided interface definition language (IDL) of Qualcomm Another IDL Compiler (QAIC). The file extension for this specification is `.idl`. Accompanying this is an implementation of `rpcmem`, which allows users to allocate physically contiguous memory and physically shared memory between the host and DSP.

Make.d Build Environment: Code for DSP kernels can be created by using GNU Make, but the documentation that accompanies the Hexagon SDK recommends using a Qualcomm-developed GNU Make library called Make.d. The goal of this library is to relieve some of the difficulty on creating a makefile from scratch. Both AArch64 and Hexagon DSP codes are supported in this library.

A general makefile is supplied as part of the Make.d build environment that can be reused across different projects, but users are responsible for supplying `.min` files that describe the flags, source and include paths, libraries to be linked, and so on specific to the user's application. The DSP and CPU are conventionally named `android.min` and `hexagon.min`, respectively. The CPU and DSP also require `android_deps.min` and `hexagon.min` files. These are responsible for listing the variants that are supported by the design (e.g., 32 bit or 64 bit ARM binaries) or a specific version of the Hexagon DSP architecture. Once each of these files is created, the `make` command must be invoked twice: once to build the Android side application code and once to build the Hexagon DSP code.

Obtaining a Handle to the DSP Domain: To target one of the DSPs available on Snapdragon, users must acquire a handle to that DSP's domain. Users must provide the universal resource identifier, which is autogenerated by the QAIC IDL tools, and a pointer to the resulting handle.

Creating Hexagon DSP Kernels: DSP kernels can be created using C, C++, or Hexagon assembly languages. For a function to be called by the host processor, users must implement a function with a signature that matches the one specified in the aforementioned `.idl` file.

It is possible to create multithreaded DSP applications, and this is done by using functionality provided by the `dspCV` library in the Hexagon DSP. Contrary to the name, this library has more functionality outside of computer vision tasks.

Another feature is the 1,024 bit vector coprocessor. This processor implements an instruction set called HVX. Although it cannot execute floating-point operations, it can execute integer and fixed-point operations. One of the difficult aspects in leveraging the coprocessor is that users must specify their design by using either assembly code or vector intrinsics. Crafting assembly code might allow users to extract the most performance possible for a given DSP design because users can craft an instruction schedule not attainable by the supplied compiler. The opposite end of this spectrum is to try to make the compiler auto-vectorize the code.

III. Cosmic Castle Framework

In §II, the architectural diversity and programming complexity of the Snapdragon SoC demonstrate the complex problem on programming heterogeneous SoCs. To address these problems, the authors propose Cosmic Castle, a framework for performance portable programming on heterogeneous DSSoCs. Cosmic Castle is a performance modeling-based, multilevel hardware-software integration strategy that will provide a methodology for projecting application ontologies onto programming systems, operating systems, and hardware. Cosmic Castle develops solutions across areas such as performance modeling and software tools to enable a development ecosystem that exercises the full capability of the highly programmable system and intelligent scheduling to manage the set of domain resources in the context of specific applications. This is an ongoing project, and this paper focuses on the main programming system of Cosmic Castle, which consists of the heterogeneous task run time system, IRIS, and its front-end compiler, OpenARC. This section introduces the IRIS runtime and OpenARC compiler and their deployment on the Snapdragon SoCs to demonstrate the potential of Cosmic Castle for heterogeneous DSSoC programming.

A. IRIS : Intelligent Run Time System

Achieving functional and performance portability in heterogeneous programming is challenging. Programmers must write the code blocks in the application to the target accelerators by using their vendor-specific programming systems (e.g., NVIDIA CUDA, AMD HIP, Intel oneAPI, Qualcomm Hexagon SDK). To fully exploit the power of all heterogeneous compute elements present in a given system, programmers must manually author and assign the code blocks/kernels to a specific accelerator and maintain them. This results in poor code performance portability and low programmer productivity.

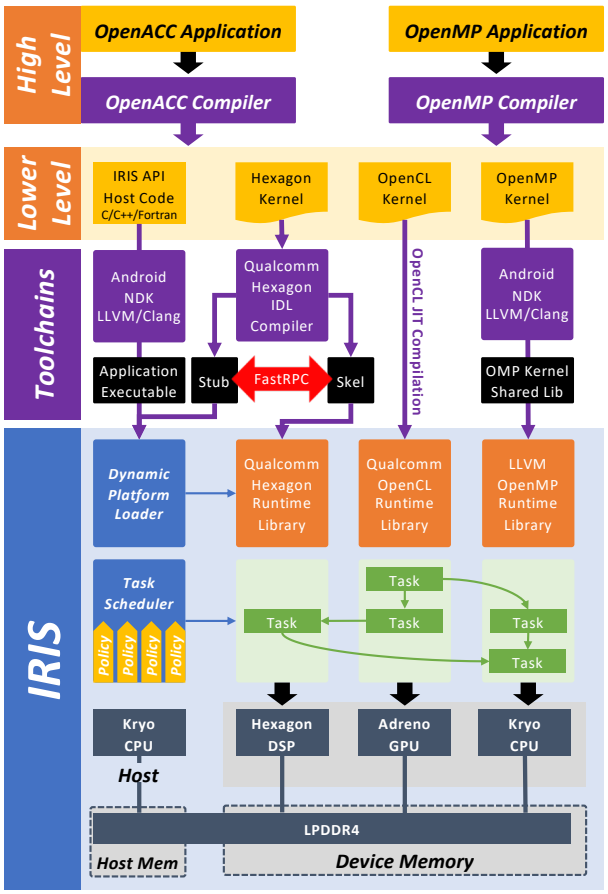

Fig. 2: IRIS for Snapdragon.

Although OpenCL was one solution proposed to address parts of this problem, not all hardware accelerators support OpenCL. As extremely heterogeneous architectures become more widespread [25], this could become a serious burden to application programmers. Also, data transfers and synchronizations among different types of accelerators remain obstacles for achieving easy programming and high performance.

To address these challenges, the authors designed and implemented IRIS, a new run time system for heterogeneous architectures [22]. IRIS dynamically catalogs and manages all the heterogeneous hardware accelerators in the system. At initialization, IRIS discovers and enumerates the heterogeneous capabilities on a node, including their preferred programming models (e.g., OpenMP, CUDA, HIP, oneAPI, Hexagon, OpenCL). IRIS enables programmers to write portable applications across diverse heterogeneous architectures.

B. IRIS on Snapdragon

For a preliminary evaluation of the IRIS framework, the authors designed and implemented IRIS for Snapdragon. Figure 2 shows the overview of IRIS for Snapdragon. As explained in §II, the target Snapdragon 855 consists of Kryo CPUs, Adreno GPU, and Hexagon DSP. Qualcomm provides the programmers with the Qualcomm OpenCL framework and Hexagon DSP SDK for their Adreno GPUs and Hexagon DSPs, respectively. IRIS uses both programming frameworks to manage the GPU and DSP devices. To target Kryo CPUs, IRIS uses OpenMP. An IRIS application for Snapdragon consists of the host code written with the IRIS C/C++/Fortran API and three native kernels: OpenMP for CPU, OpenCL for GPU, and Hexagon for DSP.

Programmers can build the application executable from the host code by using the Android Native Development Kit (NDK) LLVM/Clang compiler toolchain. The OpenMP kernel is also compiled to the OpenMP kernel shared library by using the same toolchain. The OpenCL kernel does not need to be compiled to the binary because the Qualcomm OpenCL framework for Adreno GPUs supports the Just-In-Time (JIT) OpenCL kernel compilation during the application execution. The Qualcomm Hexagon IDL compiler compiles the Hexagon kernel and builds the `skel` shared library for the DSP and the `stub` shared library for the CPU to enable Qualcomm FastRPC. The Qualcomm FastRPC allows tasks to be offloaded to DSP from CPU. The stub hides the serialization of function call parameters and the network-level communication to present a simple invocation mechanism to the application on CPU. `skel` is responsible for dispatching the call to the actual remote object implementation on DSP.

When the application runs, the dynamic platform loader in IRIS loads the Qualcomm OpenCL runtime shared library, Hexagon DSP SDK runtime shared library, and OpenMP kernel shared library at run time to execute the kernels on top of the loaded programming platforms.

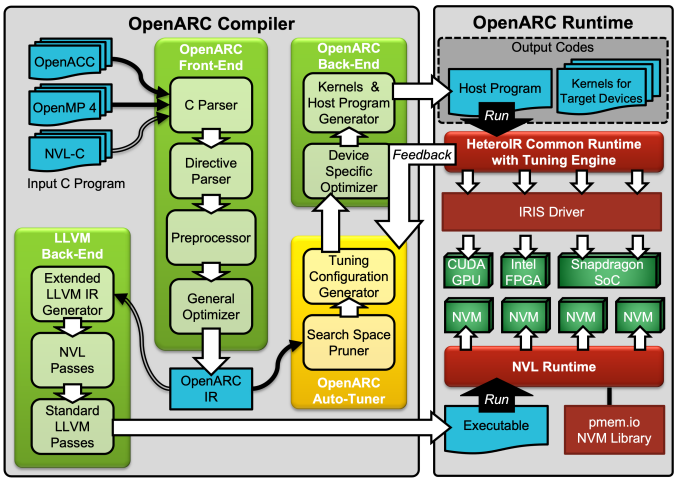

Fig. 3: OpenARC System Overview.

C. OpenARC as a Front-End IRIS Compiler

As shown in §III-B, IRIS supports multiple different target-specific native kernels (e.g., OpenMP, OpenCL, CUDA). The programmer usually authors these kernels. To automatically generate target-specific kernels from directive-based high-level programming models (e.g., OpenACC and OpenMP) for use in IRIS, the authors leveraged OpenARC [10] as the front-end IRIS compiler. OpenARC is a research compiler framework developed at Oak Ridge National Laboratory that compiles

and optimizes an input the OpenACC/OpenMP4 program for various target architectures, including CPUs, NVIDIA/AMD/Intel GPUs, Xeon Phis, Intel field-programmable gate arrays (FPGAs), and nonvolatile memory systems (e.g., Fusion-IO ioScale) [26], [27]. OpenARC offers a high-level intermediate representation and extensible annotation framework, which makes it suitable for various source-to-source translation and instrumentation studies. The high-level abstraction and various directive extensions offered by OpenARC make it easy to understand, access, and transform input programs so that the same application can be adapted differently, depending on the characteristics of target systems [28].

D. OpenARC-IRIS Integration

The original implementation of OpenARC generates multiple output programming models (e.g., CUDA, OpenCL, OpenMP, HIP), depending on the target architecture, but it only supports one back-end programming model or device type at a time. To better program and concurrently support diverse heterogeneous types of devices by using directive-based high-level programming models (e.g., OpenACC, OpenMP) where each device could require different back-end programming models (e.g., CUDA, OpenCL, OpenMP), the authors integrated OpenARC with IRIS (§ III-A). As shown in Figure 3, the new integrated OpenARC system uses IRIS as the common device runtime abstraction, which allows a single application written via a directive-based high-level programming model to exploit multiple types of devices available in the target system by automatically translating the high-level OpenACC/OpenMP4 program into multiple different back-end programming models and intermixing them (e.g., CUDA, OpenCL, OpenMP).

In the new OpenARC/IRIS framework, all types of kernel codes are generated during one invocation of the compiler while still allowing for the application of various architecture-specific optimizations, such as: (1) exploiting device-specific memories (e.g., CUDA shared/texture memory); (2) reduction optimization to exploit FPGA-specific special hardware mechanisms, such as shift registers; and (3) sliding window optimization to avoid redundant memory accesses across loop iterations and hide memory latency using hardware buffers [11], [29]. The authors also optimized the OpenARC/IRIS interface in which an additional compiler pass automatically merges IRIS tasks that belong to the same OpenACC construct. Additionally, the new optional runtime API can be used to automatically merge multiple IRIS tasks across different OpenACC constructs if users guarantee its safety. This can reduce OpenARC/IRIS interfacing overheads.

IV. Evaluation

This section evaluates the effectiveness of the proposed Cosmic Castle programming system by porting benchmarks from different application domains to heterogeneous compute units available in the Snapdragon SoC and comparing their performance against other traditional HPC devices (e.g., Intel Xeon E5-2683 CPU, AMD EPYC 7742 CPU, ARM ThunderX2 CPU, NVIDIA P100 GPU). The tested device-specific kernels (e.g., OpenMP, OpenCL, CUDA), except for the hand-optimized versions, are automatically generated by OpenARC from the corresponding input OpenACC programs.

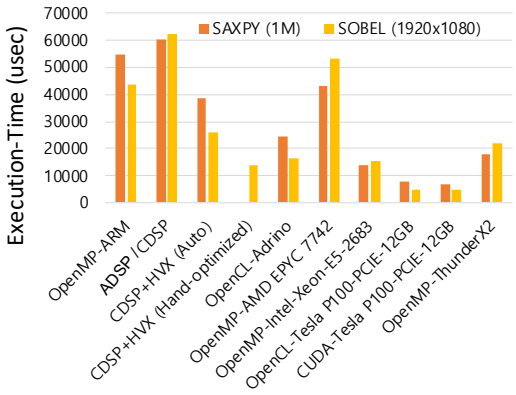

Fig. 4: SAXPY and Sobel filter benchmarking.

A. Performance Comparison of Heterogeneous Compute Units in Snapdragon SoC

The authors benchmarked SAXPY and Sobel filter kernels to compare the effectiveness of Snapdragon compute units when compared with Intel, AMD, ThunderX2, and NVIDIA compute resources, and the results are shown in Figure 4. The execution time includes kernel execution time and IRIS runtime overhead. It was also compared with HPC CPU and GPU computing resources. The AMD EPYC 7742 system has two EPYC 7742 processors, and each processor has 64 cores with simultaneous multithreading enabled and 1 TB of main memory running at a maximum frequency of 2.25 GHz. The Intel Xeon E5-2683 system has 32 cores in two sockets running at a maximum frequency of 3 GHz and has 256 GB of main memory. The ARM ThunderX2 system has two CPUs, each with 28 cores with 4 threads per core. ThunderX2 has 128 GB of main memory and is running at a maximum frequency of 2.5 GHz.

The authors listed the programming model (i.e., OpenMP, OpenCL, and CUDA) used for each compute resource. For CDSP, the kernel is written in C/C++ and is interfaced through Hexagon FastRPC. The Hexagon CDSP without HVX results are 43% worse than those of OpenMP-ARM (with NEON through auto-vectorization) optimizations. The authors observed similar results with ADSP of the Snapdragon board. The HVX instructions can be generated through the auto-vectorization feature, and the results are 40% better than those of OpenMP-ARM. The hand-optimized HVX code running on CDSP (shown only for the Sobel filter under CDSP+HVX/hand-optimized) results in a 3.2× gain compared with OpenMP-ARM. The OpenCL code running on the Adreno GPU of the Snapdragon SoC also results in performance similar to that of CDSP with hand-optimized HVX code. The effectiveness of Snapdragon computing resources were compared with NVIDIA, Intel, AMD, and ThunderX2 computing resources. Snapdragon Adreno/CDSP resources are 3× worse than NVIDIA Tesla

P100 GPU results. However, its performance is comparable with that of ThunderX2.

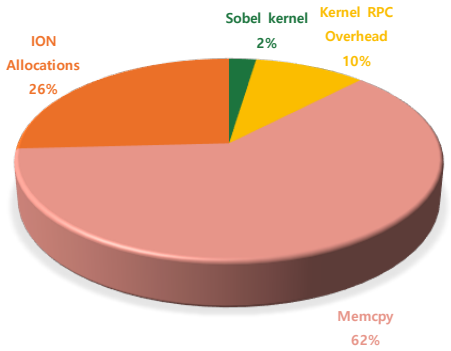

Fig. 5: Sobel filter run on hexagon DSP with run-time split.

The authors further analyzed the execution time of the Sobel filter when executing on the Hexagon DSP. The results of this analysis are shown in Figure 5. The actual Sobel filter kernel on the Hexagon CDSP takes only 2% of the overall execution time. The remaining time is consumed by FastRPC, ION memory allocations, and data transfers to/from ION memory. Hence, there is ample opportunity for run time management of kernels to optimize the data transfers and ION memory allocations by making the kernels efficiently use the memory and I/O.

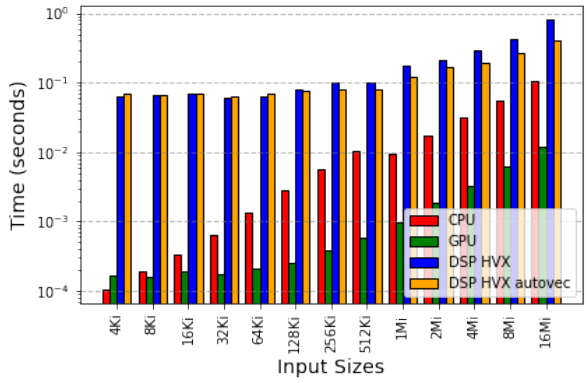

Fig. 6: Comparison of CPU, GPU, DSP, and DSP auto-vectorized IAXPY.

B. Hexagon DSP Kernel Performance

To further analyze the performance characteristics of the Hexagon DSP in the Snapdragon SoC, the authors also ported the IAXPY (integer $a \times x + y$) kernel by using two different implementation approaches: one with and one without auto-vectorization. Disassembling the two versions shows that the auto-vectorized disassembly has almost 10× as many instructions. However, Figure 6 shows that up to 2× as much performance can be achieved by invoking the auto-vectorizer. Although the results show that the CPU and GPU perform these computations faster than the DSP, there is a savings in power consumption that is not displayed in this graph. Although there is not a direct way to measure power consumption of the Hexagon DSP, the authors plan to estimate power consumption by architecting a performance model in future work.

C. IRIS Performance Portability on Snapdragon

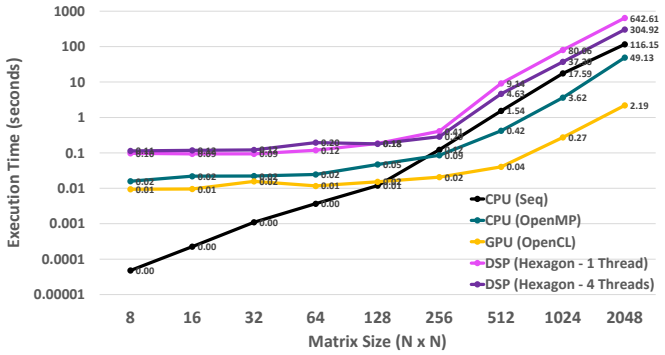

Fig. 7: SGEMM on Snapdragon.

To show the performance and portability of IRIS on Snapdragon, the authors evaluated Single Float Precision General Matrix Multiply (SGEMM) on the Snapdragon SoC, as shown in Figure 7. The SGEMM application contains host code written in the IRIS C API and three different kernels, including an OpenMP kernel for CPU, OpenCL kernel for GPU, and Hexagon kernel for DSP. For the Hexagon kernel, the number of running threads were varied from 1 to 4. The sizes of the matrices in the application were varied from 8 × 8 to 2,048 × 2,048 in powers of two. For the baseline performance, a sequential version of SGEMM running on a single CPU core only was also evaluated. As the size of the matrices increases, GPU shows the best performance, and DSP show the worst performance. This is because SGEMM is a compute-intensive kernel and good for data parallel-friendly devices (e.g., GPUs). On the other hand, DSP shows the worst performance because of its heavily pipelined hardware design and a small number of concurrent hardware threads.

V. Conclusion

Contemporary heterogeneous systems provide mechanisms to manage different types of accelerators in heterogeneous architectures. However, achieving functional and performance portability in heterogeneous programming is a challenging problem. This paper presents Cosmic Castle, a software ecosystem for performance portable programming on the heterogeneous DSSoCs. Cosmic Castle is an on-going project, and this paper provides a preliminary evaluation of the Cosmic Castle programming toolchain, which comprises the smart compiler OpenARC and the intelligent runtime system IRIS, by porting and evaluating various benchmarks on the Snapdragon SoC as an example heterogeneous DSSoC. The results show that Cosmic Castle allows users to program DSSoCs by using directive-based high-level programming models while exploiting and intermixing different device-specific programming models preferred by each heterogeneous device. However, the initial performance comparison against manual low-level implementations and other traditional HPC devices shows the need for exploring further optimization opportunities to achieve better performance portability.

Acknowledgments

This research used resources of the Experimental Computing Laboratory (ExCL) at Oak Ridge National Laboratory, which is supported by the US Department of Energy (DOE) Office of Science under contract no. DE-AC05-00OR22725.

This research was supported by the following sources: (1) the Defense Advanced Research Projects Agency Microsystems Technology Office Domain-Specific System-on-Chip Program and (2) DOE Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing program.

References

[1] OpenACC, "OpenACC: Directives for accelerators," 2015.

[2] OpenMP, "OpenMP reference," 1999.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[4] Kokkos, "The C++ Performance Portability Programming Model," [Online]. Available: https://github.com/kokkos/kokkos, 2014.

[5] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[6] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Understanding portability of a high-level programming model on contemporary heterogeneous architectures," *Micro, IEEE*, vol. 35, no. 4, pp. 48–58, 2015.

[7] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*. Salt Lake City, Utah, USA: IEEE press, 2012.

[8] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with xeon phi KNL," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 297–306.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *ACM Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. Pittsburgh, Pennsylvania: ACM, 2010, pp. 63–74.

[10] S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Vancouver: ACM, 2014.

[11] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Chicago: IEEE, 2016.

[12] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Raleigh, NC, USA: ACM, 2009.

[13] S. Lee and R. Eigenmann, "OpenMPC: Extended openMP programming and tuning for GPUs," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[14] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, *Pragmatic Performance Portability with OpenMP 4.x*. Cham: Springer International Publishing, October 2016, pp. 253–267. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-45550-1_18

[15] cuBLAS, "Dense Linear Algebra on GPUs," [Online]. Available: https://developer.nvidia.com/cublas, 2012.

[16] Thrust, "A Parallel Algorithm Library," [Online]. Available: https://developer.nvidia.com/thrust, 2012.

[17] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 11:1–11:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063398

[18] C. LengauerSven, A. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "Exastencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops*, 2014, pp. 553–564.

[19] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Korner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 210–224, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2015.2394802

[20] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601174

[21] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 85:1–85:11, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2897824.2925892

[22] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "IRIS: A portable runtime system exploiting multiple heterogeneous programming systems," in *IEEE High Performance Extreme Computing (HPEC)*, 2021.

[23] Intel, "Intel Intrinsics Guide," [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[24] ARM, "ARM Developer: Neon Intrinsics Reference," [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics.

[25] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, A. Dubey, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, J. P. Peltz, T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018: DOE ASCR basic research needs workshop on extreme heterogeneity," 2018.

[26] J. E. Denny, S. Lee, and J. S. Vetter, "NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. Kyoto, Japan: ACM, 2016, pp. 125–136.

[27] S. Lee and J. S. Vetter, "OpenARC: extensible openACC compiler framework for directive-based accelerator programming study," in *Proceedings of the First Workshop on Accelerator Programming using Directives (with SC14)*. New Orleans: IEEE Press, 2014, pp. 1–11.

[28] J. Lambert, S. Lee, J. Vetter, and A. Malony, "CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 1387–1400. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00102

[29] J. Lambert, S. Lee, J. Kim, J. S. Vetter, and A. D. Malony, "Directive-based, high-level programming and optimizations for high-performance computing with FPGAs," in *ACM International Conference on Supercomputing (ICS)*. Beijing: ACM, 2018.