On the Portability of GPU-Accelerated Applications via Automated Source-to-Source Translation

Paul Sathre
Virginia Tech
Dept. of Computer Science
Blacksburg, Virginia, USA
sath6220@vt.edu

Mark Gardner
Virginia Tech
Dept. of Computer Science
and the Office of IT
Blacksburg, Virginia, USA
mkg@vt.edu

Wu-chun Feng
Virginia Tech
Dept. of Computer Science
Dept. of Elec. & Computer Engg.
Blacksburg, Virginia, USA
wfeng@vt.edu

ABSTRACT

Over the past decade, accelerator-based supercomputers have grown from 0% to 42% performance share on the TOP500. Ideally, GPU-accelerated code on such systems should be "write once, run anywhere," regardless of the GPU device (or for that matter, any parallel device, e.g., CPU or FPGA). In practice, however, portability can be significantly more limited due to the sheer volume of code implemented in non-portable languages. For example, the tremendous success of CUDA, as evidenced by the vast cornucopia of CUDA-accelerated applications, makes it infeasible to manually rewrite all these applications to achieve portability. Consequently, we achieve portability by using our automated CUDA-to-OpenCL source-to-source translator called CU2CL. To demonstrate the state of the practice, we use CU2CL to automatically translate three medium-to-large, CUDA-optimized codes to OpenCL, thus enabling the codes to run on other GPU-accelerated systems (as well as CPU- or FPGA-based systems). These automatically translated codes deliver performance portability, including as much as *three-fold* performance improvement, on a GPU device not supported by CUDA.

ACM Reference Format:
Paul Sathre, Mark Gardner, and Wu-chun Feng. 2019. On the Portability of GPU-Accelerated Applications via Automated Source-to-Source Translation. In *Proceedings of International Conference on High Performance Computing in Asia Pacific Region (HPC Asia'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The desire for higher-fidelity simulation and data-driven scientific computation has long been a key driver behind high-performance computing (HPC). Such scientific computation was achieved via parallelism at both the intra- and inter-node scales on relatively homogeneous hardware, typically via MPI+OpenMP or MPI alone. Rather than relying on an increasing number of heavyweight CPU cores, the HPC community began pairing these CPU cores with additional accelerators (or coprocessors), onto which all or a portion of key computations could be offloaded to take advantage of the performance afforded by their radically different architectures. This ongoing transition is evidenced by the increasing performance share that accelerators have within the top performing supercomputers in the world. Between November 2005 and November 2018, their combined share grew from 0% to nearly 42% of all performance on the Top500 List [19]. Though many varieties of accelerator have been used and new types are being developed, the most commonplace accelerator is the graphics processing unit (GPU). Despite their prevalence, GPUs are still non-trivial architectures to develop code for. Their dissimilarity to traditional CPUs requires adapting to a new mental model of parallel execution. The more significant obstacle, however, is the lack of a common programming abstraction, language, and runtime.

The first general-purpose GPU (GPGPU) programming abstraction to gain significant traction was NVIDIA's Compute Unified Device Architecture (CUDA), which provides a single-source approach to programming both host (CPU-side) and device (GPU-side) code and coordinating between the two. A large body of CUDA-based accelerated software has amassed since its inception in 2007, largely due to the convenient programming abstraction it provides. However, as a vendor-owned language and runtime specification, CUDA code can only execute on NVIDIA GPUs without the use of third-party tools.

Approximately two years after CUDA emerged, efforts to create a vendor-neutral, standard approach to programming GPUs and other parallel platforms culminated in the Open Compute Language (OpenCL). OpenCL provides a similar programming model to CUDA, albeit with more cumbersome syntax (particularly on the host side); but it also delivers functional portability (similar to C) across any parallel computing device that provides a compliant implementation of OpenCL.

CUDA and OpenCL have similar programming models, hence, there is an opportunity to extend the reach of existing CUDA codes to the additional platforms accessible via OpenCL. However, manually translating the millions of lines of CUDA code already written would be intractable, necessitating a huge developer effort with a high risk of human-introduced errors. Therefore, we leverage our automated CUDA-to-OpenCL source-to-source translator (CU2CL), available at [1] and discussed further in §3, to explore the portability of optimized CUDA codes to devices from different vendors. In particular, we compare the portability of three real-world accelerated applications: (1) lid-driven cavity (*LDC*), a canonical problem in computational fluid dynamics (CFD), (2) *Fen Zi*, a molecular dynamics simulation, and (3) *GEM*, a molecular modeling application. All of

This work was supported in part by NSF I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing.

HPC Asia'19, January 14–16, 2019, Guangzhou, China

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

these applications are discussed in greater detail in §4. We discuss other approaches to portability and related work in §5.

Specifically, we make the following contributions:

- Demonstration of the practicality of running CUDA codes on non-CUDA devices via static source-to-source translation
- Verification that the performance and correctness of translated CUDA applications is preserved on CUDA devices and may even improve on non-CUDA GPUs
- An analysis of the functional and performance portability of CUDA-optimized codes on non-GPU devices, specifically Intel Xeon Phi (MIC), Intel CPU, and Altera FPGA
- Identification of optimizations to improve the performance portability of codes targeting Intel Xeon Phi

2 BACKGROUND

For a given code sub-expression to be portable across GPU languages, a semantically-equivalent expression must be composable in the target language. In general, CUDA and OpenCL possess this property, i.e., device-side computations are expressed as *kernel functions* that are invoked from the host (CPU); data is mapped or explicitly copied to/from the device (GPU); and compute threads and memory are organized into comparable hierarchies. The kernel languages are similar, primarily differentiated by syntax, as shown in Table 1. This similarity has led to several efforts to facilitate portability between the two languages, as discussed in §5.

CUDA	OpenCL	Purpose
__global__	__kernel	declare a host-invokable device function
*type**	__global *type**	declare a kernel parameter residing in the device global memory space
blockIdx.*{x,y,z}*	get_group_id(*{0,1,2}*)	query block/workgroup index
blockDim.*{x,y,z}*	get_local_size(*{0,1,2}*)	query block/workgroup size
threadIdx.*{x,y,z}*	get_local_id(*{0,1,2}*)	query in-block/in-group index

Table 1: OpenCL equivalents for a few syntactic elements.

CUDA evolves to include features as soon as they are supported by new NVIDIA devices, whereas OpenCL's feature set lags because of the need for support *across* multiple heterogeneous devices, ranging from server to desktop to embedded. The OpenCL standard continues to evolve and the 2.x specifications include features that were once exclusive to CUDA, such as a shared virtual memory address space, device-side kernel enqueueing (i.e., *dynamic parallelism*), and device-side C++ [14, 22]. Thus, CUDA codes have become increasingly easier to translate as the OpenCL standard continues to evolve.

Performance is another concern when porting codes from CUDA to OpenCL. Accelerators from different vendors released around the same time are expected to achieve similar levels of performance. This healthy competition is why having tools to enhance portability are so important. Portability allows devices to be procured on the basis of performance, power consumption, and cost rather than limiting the decision to only language compatibility.

Initially, the optimizations needed to deliver performance portability differed between NVIDIA and AMD GPUs. What ran well on one would often run poorly on the other. However, with the AMD GPU having moved from a VLIW architecture to a scalar architecture in the early 2010s, the optimizations needed for both are now very similar [7]. With the convergence in GPU architecture, we find it possible to achieve not only *functional portability* but also *performance portability* across GPU vendors. Additionally, non-GPU accelerator devices such as the Intel Xeon Phi and FPGAs have captured a significant market and performance share as viable alternative devices. In this paper, we explore the potential to achieve functional and performance portability across these devices by leveraging the CU2CL translator by Gardner et al. [12] to translate and evaluate three scientific applications, as noted earlier: (1) lid-driven cavity (LDC), a computational fluid dynamics (CFD) application, (2) *Fen Zi*, a molecular dynamics application, and (3) GEM, a molecular modeling application.

3 CU2CL TRANSLATOR

The CU2CL tool [12] is an automated CUDA-to-OpenCL source-to-source translator (CU2CL), enabling portability of CUDA software to platforms lacking a CUDA implementation. We chose OpenCL as the target language for the following reasons: (1) its broad-based adoption and support, e.g., it currently supports not only GPUs but also CPUs, Intel Xeon Phi, and even FPGAs, (2) a similar programming model to CUDA, and (3) a "write-once, run-anywhere" open standard. While CUDA and OpenCL are quite similar, the syntactic changes required to decouple CUDA's unified host and device source files require the contextual information provided by a full compiler framework. For this reason, CU2CL builds upon the open-source Clang compiler, which in turn builds upon the LLVM compiler infrastructure. Figure 1 provides a conceptual overview of the translation pipeline and the core Clang components CU2CL utilizes.

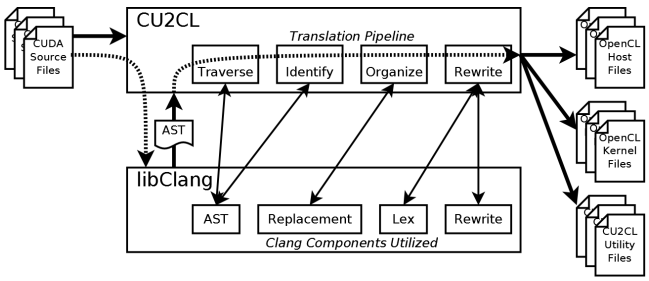

Figure 1: Architecture of CU2CL and its relationship to Clang.

CU2CL performs *AST-driven,*[1] *string-based* translation locally and synthesizes global translations by sharing state between the multiple ASTs that make up a full binary. AST-driven, string-based translation means that CU2CL walks the AST generated by Clang to identify critical source components and performs translations by replacing the relevant text strings directly rather than by transforming and flattening the AST. This provides the benefit of retaining the full context in the generated OpenCL source files, preserving preprocessor directives, commenting, and formatting. AST-driven translations are performed in a depth-first manner, where leaf nodes of the AST are first translated, and then any edited code is injected in

[1] AST: abstract syntax tree

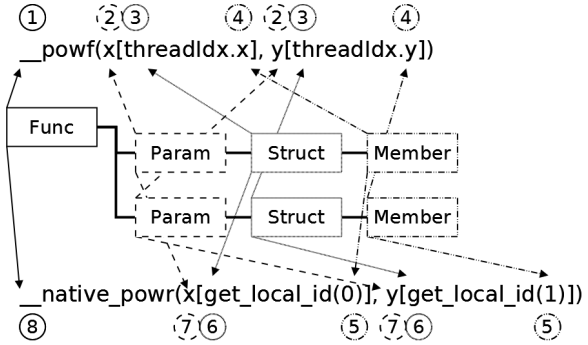

Figure 2: AST-driven, syntax-directed translation with CU2CL.

the wrapping statement. Figure 2 shows an example of this process by translating a kernel-intrinsic function call in eight stages: (1) find a CUDA construct to translate, (2) recognize the function arguments as array parameters, (3) recognize that the indices into the arrays are `threadIdx` structs ..., (4) ... with struct members as the indices, (5) translate `threadIdx` members to the appropriate parameter for OpenCL's `get_local_id` function call, (6) translate the `threadIdx` structs to `get_local_id` function calls, (7) pass the standard C array references unchanged, and (8) translate the CUDA `__powf` function to the OpenCL equivalent `native_powr`.

4 PORTABILITY THROUGH TRANSLATION

In this section, we demonstrate the efficacy of automated source-to-source static translation via CU2CL to provide portability and performance on non-CUDA devices via three scientific applications: (1) lid-driven cavity (*LDC*), a canonical problem in computational fluid dynamics (CFD), (2) *Fen Zi*, a molecular dynamics simulation, and (3) *GEM*, a molecular modeling application, as shown in Figure **??**.

4.1 The Lid-Driven Cavity (LDC) Simulation

The LDC application is a canonical simulation in computational fluid dynamics (CFD), which simulates viscous incompressible fluid flow in a square cavity with a moving boundary on only one side. The CUDA simulation encompasses eight distinct kernels using 5-point and 9-point stencils over a two-dimensional (2D) simulation grid. Our simulation set-up for LDC uses the same experimental parameters as found in [20]: fluid density of 1 kg/m^3, Reynolds number of 100, and lid velocity of 1 m/s computing for exactly 1000 iterations. The computational grid varied from 128^2 to 4096^2. We conducted all analyses on the machine configurations shown in Table 2, where all the accelerator devices come from a similar time frame (2015). Auto-translated OpenCL running on both NVIDIA and AMD GPUs produced *exactly* the same residual output at every 50-timestep interval as the original CUDA. Running on the CPU and Xeon Phi, the residuals differed due to floating-point roundoff error that is caused by a reordering of the reduction operation. However, this reduction was only used to compute the residual diagnostic and not used by the simulation itself.

The performance of the LDC benchmark across a range of grid sizes for each of the GPU platforms, shown in Fig. 3, provides some interesting insights about the relative strengths of each of the compute devices.[2] For the smaller grid sizes (i.e., 128^2 and 256^2 elements),

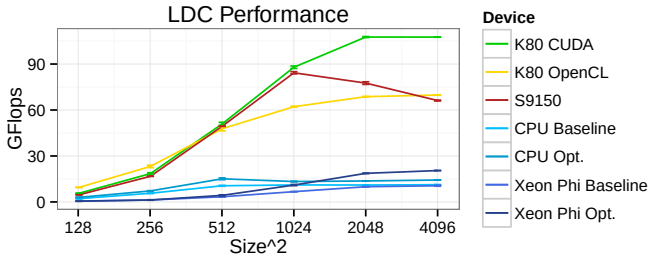

Figure 3: Reported GFlops achieved within the iterative loop of the LDC simulation as a function of problem size and compute device.

our auto-translated OpenCL code performs *better* on the NVIDIA GPU device than the original CUDA code does on that same NVIDIA device. This suggests that the kernel-launch overhead for the CUDA implementation is higher than for the OpenCL implementation since that overhead is more pronounced at smaller grid sizes (where there is insufficient work to amortize the cost of the launch overhead). However, as the problem scales to realistic work sizes, the efficiency of NVIDIA's CUDA implementation far surpasses that of NVIDIA's OpenCL implementation. At the largest problem size, the NVIDIA OpenCL implementation only achieves 65% of the performance of the original NVIDIA CUDA implementation.

For the AMD S9150 GPU, the performance of our auto-translated OpenCL tracks the performance of the original CUDA performance through 1024^2 grid elements. After this point, however, the performance of the AMD S9150 GPU on LDC *degrades*. This degradation is not due to a limitation of GPU memory, as the working set size for the simulation is 8 bytes/variable × 3 variables/cell × ($xdim \times ydim$) cells. This translates to approximately 400MB for the 4096^2 problem size vs. the 16GB of available GPU memory. So, what causes the performance degradation? As shown in later sections, this degradation at scale turns out to be unique to LDC and the AMD S9150 GPU, where we observe that during a critical phase of the LDC computation, and on the S9150 only, the CPU reaches 100% utilization running the ksoftirqd daemon.

For the Intel Xeon Phi (Knights Corner) platform, the performance of our auto-translated OpenCL implementation is underwhelming.[3] Hotspot analysis with Intel Vtune indicates that the majority of the time in the two most expensive kernels, `ldc_explicit_iter` and `ldc_explicit_init_resid_output`, is spent performing unaligned loads and stores (with respect to the 512-bit vector width) to/from global memory. This prevents the full utilization of the memory bandwidth of the co-processor, and in turn, significantly impacts performance.

[2] The reported GFlops measurement takes into account device allocations, memory transfers, and kernel executions; one-time runtime initialization costs are triggered before the timed loop.

[3] In auto-translating the CUDA code to OpenCL, CU2CL also auto-translates the NVIDIA-specific CUDA optimizations, which, while largely appropriate for the AMD S9150 GPU, are *not* for the Intel Xeon Phi.

Platform Tested	CPU	RAM	OS / Kernel	Accelerator / Driver	Compiler(s)
NVIDIA GPU (CUDA/OpenCL) AMD GPU (OpenCL)	(2x) Intel Xeon E5-2637 v4 @ 3.50 GHz	(8x) 32 GiB DDR4 @ 2400 MHz	Debian Jessie (8.6) 3.16.0-4	Nvidia K80 (367.35) AMD S9150 (fglrx 15.30.3)	nvcc 7.5.17 gcc 4.9.2
Intel CPU (OpenCL) Intel Xeon Phi (OpenCL)	(2x) Intel Xeon E5-2697 v2 @ 2.70 GHz	(8x) 8 GiB DDR3 @ 1333 MHz	Centos 6.8 (2.6.32-642.6.2.el6.x86_64)	Intel MIC SC7120P (KNC) (MPSS 3.3.3 OpenCL 14.2)	gcc 4.4.7
Altera FPGA (OpenCL)	(2x) AMD Opteron 6272 @ 1.4 GHz	(16x) 4 GiB DDR3 @ 1600 MHz	Centos 6.8 (2.6.32-573.18.1.el6.x86_64)	Bittware S5-PCIe-HQ-D8 Stratix V (Quartus 16.0)	gcc 4.4.7

Table 2: Test machine configurations

The unaligned loads and stores occur because the LDC algorithm is designed to accept *any* size of x and y dimension and uses conditionals to mask off no-op threads in the Grid/ND-range that extend beyond the compute region during kernel execution. While this makes the algorithm very general, it prevents the compiler from guaranteeing proper load and store alignment, which then adversely impacts performance. While better performance on fixed power-of-two problem sizes can be obtained by modifying the algorithm, the goal of this study is to compare the performance of existing codes rather than optimizing for best performance on a per-platform basis. That being said, however, we did identify a few optimizations that could be incorporated into a static source-to-source translator (like CU2CL) to provide a *two-fold* performance improvement on Xeon Phi. First, several LDC kernels make use of the pow function to compute squared or cubed values; VTune indicated a non-trivial time cost for these operations. Thus, any call to pow having a constant exponent was "unrolled" to a sequence of multiplies, which could easily be added as a translation-time option. Second, despite the necessarily unaligned loading semantics, we find that the compiler does not perform sufficiently aggressive automatic prefetching. Thus, we manually inserted prefetching, first by reading values into the L2 caches that would be shared by multiple threads and later by reading thread-specific data. A source-to-source translator like CU2CL could identify critical read accesses from global memory and optionally inject appropriate prefetching code ahead of the read step. (Note: Currently, these optimizations are not included in our CU2CL translator but could be rolled in as part of a future release.)

CPU performance for a highly SIMD-izable application like LDC is not expected to be competitive with accelerator devices, but the results are included for completeness here. A slight performance gain is observed from the optimizations manually applied to the Xeon Phi, indicative of the architectures' parallel design and optimization.

4.2 The Fen Zi Molecular Dynamics Simulation

Fen Zi, a large-scale molecular dynamics simulation developed at the University of Delaware, contains 17,768 lines of CUDA code, not counting the lines of standard C++. It leverages both N-body and spectral methods and heavily utilizes the cuFFT library. The application was optimized for the NVIDIA Fermi and Kepler architectures and exploits both constant and texture memory spaces throughout the hand-written N-body kernels for efficient access to program-wide variables and arrays.

To evaluate the correctness and performance of the translated Fen Zi application with respect to the original CUDA, we used two different simulation sizes of a dimyristoyl phosphatidylcholine (DMPC) bilayer with a timestep of one femtosecond, as per [25]. The small test (DMPC Small) consists of 17,004 atoms with 14,096 bonds, 19,108 angles, and 22,536 dihedrals; it runs for 10,000 times steps. The medium test (DMPC Medium) is an approximately 2x2 expansion of the small test and consists of 68,484 atoms with 56,696 bonds, 76,588 angles, and 90,144 dihedrals; it runs for 1,000 time steps. Due to the inherent stochasticity of the molecular dynamics simulation, the results vary slightly between different platforms or different runs on the same platform. To ensure that the CU2CL translation did not change the algorithm, we performed 10 runs of both problem sizes on each platform, averaged the potential energy at each timestep, and plotted a ribbon of the 95% confidence interval around the average in Fig. 4. While the evolution varies across devices and individual runs on a single device, the trajectory of the stochastic simulation remains consistent across platforms and iteratons.[4]

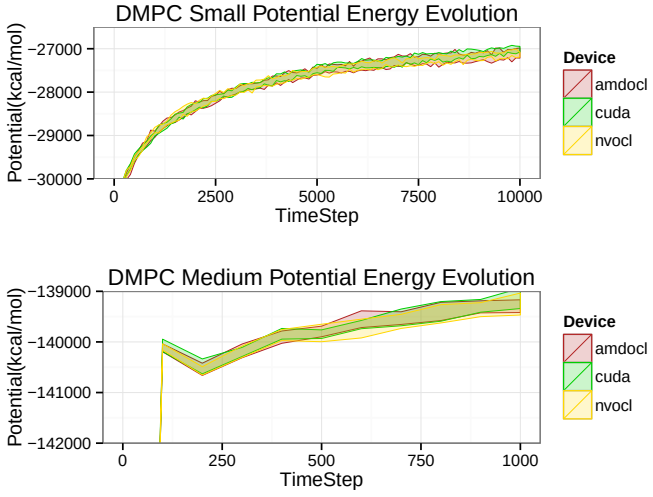

Figure 4: Comparison of the potential energy evolution of the stochastic Fen Zi simulation.

Similar to the LDC observations in §4.1, the NVIDIA OpenCL implementation only achieves 67% and 68% of the original CUDA implementation on the small and medium test problems, respectively.

[4]The auto-translated OpenCL could not be evaluated on Xeon Phi, due to its lack of image memory support to replace CUDA textures. It also could not be evaluated on the Intel CPU, due to a segmentation fault within the Intel OpenCL implementation when JIT-compiling the clFFT kernels.

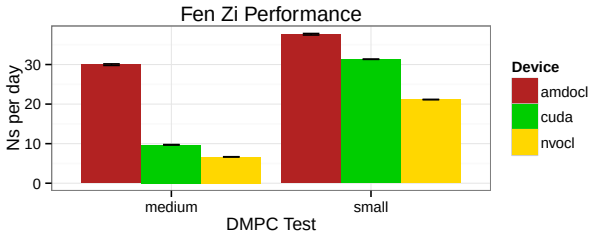

Figure 5: A comparison of nanoseconds of simulation time performed per day (reported by Fen Zi).

However, without any manual optimization, the Fen Zi OpenCL implementation (generated by CU2CL) on the AMD S9150 GPU outperforms the original CUDA implementation on the NVIDIA K80 GPU by 1.2× on the small test and 3.1× on the medium test, as shown in Fig. 5. Though somewhat counter-intuitive, CUDA-optimized applications are capable of running on comparable non-NVIDIA platforms with higher absolute performance than on the original target hardware. Previous work on AMD GPUs with VLIW-based architectures indicated that OpenCL code tuned for CUDA devices often performs worse unless specifically re-tuned [5, 17]. Today, however, the optimizations needed to improve performance on NVIDIA devices also perform well on contemporary AMD devices (i.e., after 2012, when AMD switched from a VLIW architecture to a scalar architecture with the release of the Graphics Core Next (GCN) architecture). For Fen Zi, not only does the auto-translated CUDA application faster on the AMD GPU, the performance scales better to larger problems sizes on the AMD GPU as well. When the problem size increases by roughly four-fold, the AMD GPU still performs relatively close to the same amount of simulation per day (i.e., only a 1.3× reduction) while the CUDA and OpenCL implementations on the NVIDIA GPU result in a more significant 3.2× reduction.

4.3 The GEM Molecular Modeling Simulation

GEM [10, 13] is a molecular-modeling application that computes the electrostatic surface potential of a biomolecule using an N-body computation. It has been manually ported to several GPU and other accelerator runtimes and platforms over the years, including ATI Brook+ [3], NVIDIA CUDA [15], OpenCL [5], and Intel MIC via OpenMP and AVX intrinsics [18]. The N-body kernel is known to be amenable to both multi- and many-core architectures. It computes the surface potential at M vertices along a surface, based on a sum of contributions from individual charges at each of N atoms in the biomolecule. Specifically, the potential is calculated based on the analytic, linearized Poisson-Boltzmann model.

The Kepler-optimized version of GEM uses a single non-branching kernel to compute all atomic-charge contributions. Each thread computes a single vertex from all atoms in the molecule. Within the kernel, vertex data is manually cached in `float2` shared-memory buffers for in-thread reuse, and atom data is exchanged within a warp via the CUDA `__shfl` intrinsic that allows in-register sharing. However, because OpenCL exposes no such register-swap intrinsic, this must be emulated via local memory.[5] The emulation

[5] AMD GPUs have a corresponding intrinsic that is not yet exposed via OpenCL.

```
atom_xloc=__shfl(atom_xloc, copyfrom, 32);
```

(a) Example usage of the CUDA `__shfl` intrinsic.

```
shfl_stage[tid]=atom_xloc;
barrier(CLK_LOCAL_MEM_FENCE);
atom_xloc=shfl_stage[copyfrom];
barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Example of locally-staged `__shfl` emulation in OpenCL.

Figure 6: Emulating the CUDA `__shfl` intrinsic in OpenCL.

uses $sizeof(DataType) \times WorkGroupSize$ additional shared memory and two workgroup-level synchronizations (one for writing and one for reading), which dramatically increases the exchange cost due to the increased latency of local memory operations and synchronization. Code for the emulation is provided in Fig. 6.

Molecule Short Name	Molecule Full Name	PDB ID	Atoms	Vertices
Mb.HHelix [21]	Myoglobin: H Helix	1MBO	382	5884
1uwo_A [23]	Calcium Form of Human S100B: Chain A	1UWO	1441	16529
1qks_A [11]	Cytochrome CD1 Nitrite Reductase, Oxidised form: Chain A	1QKS	8542	58018
nucleosome [6]	Nucleosome Core Particle	1KX5	25086	258797
2eu1 [4]	Chaperonin GroEL-E461K Crystal Structure	2EU1	109802	898584
capsid [26]	Tobacco Ringspot Viral Capsid	1A6C	476040	593615

Table 3: Macro-molecules used as input.

The original code and translated code are executed with six progressively larger biomolecules as input. Tab. 3 provides the relevant protein database (PDB) identifiers and atom/vertex counts for these biomolecules. The correctness of the computed surface potential is validated on both platforms by a root mean square (RMS) error analysis, as in [3]. We calculate the RMS error and normalized RMS error for the computed surface potential across all vertices for each of the six tested biomolecules by dividing by the difference between the maximum and minimum vertex potentials to account for the magnitude of the vertex charge and compare against the CUDA version. The computed errors in Tab. 4a for both NVIDIA OpenCL and AMD OpenCL are well within the expected for the minimal reordering of charge accumulation computations that occur when switching between architectures with the same threading model. The Xeon Phi, CPU, and Altera platforms cannot take advantage of the atom shuffle behavior of the GPUs due to compilation issues with barriers inside kernel loops. Therefore, we used a simpler loop, where all vertices accumulate from the same atom at the same time, which reorders the accumulations and results in the larger floating-point roundoff error on non-GPU devices, as shown in Tab. 4b.

Molecule Short Name	NVIDIA K80		AMD S9150	
	RMS	Normalized RMS	RMS	Normalized RMS s
Mb.HHelix	1.1×10^{-5}	1.4×10^{-6}	4.4×10^{-5}	5.6×10^{-6}
1uwo_A	1.7×10^{-5}	1.6×10^{-6}	1.2×10^{-4}	1.1×10^{-5}
1qks_A	2.4×10^{-5}	2.3×10^{-6}	2.0×10^{-4}	1.9×10^{-5}
nucleosome	4.1×10^{-5}	4.2×10^{-6}	1.6×10^{-3}	4.8×10^{-5}
2eu1	5.0×10^{-5}	2.5×10^{-6}	2.9×10^{-4}	1.5×10^{-5}
capsid	5.1×10^{-5}	7.0×10^{-6}	1.0×10^{-4}	1.5×10^{-5}

(a) RMS and normalized RMS for auto-translated OpenCL on GPU devices.

Intel CPU		Intel MIC		Altera FPGA	
RMS	Normalized RMS	RMS	Normalized RMS	RMS	Normalized RMS
7.0×10^{-1}	8.9×10^{-2}	7.0×10^{-1}	8.9×10^{-2}	7.0×10^{-1}	8.9×10^{-2}
5.5×10^{-1}	5.4×10^{-2}	5.5×10^{-1}	5.4×10^{-2}	5.5×10^{-1}	5.4×10^{-2}
3.8×10^{-1}	3.8×10^{-2}	3.8×10^{-1}	3.8×10^{-2}	3.8×10^{-1}	3.8×10^{-2}
4.8×10^{-1}	1.5×10^{-2}	4.8×10^{-1}	1.5×10^{-2}	4.8×10^{-1}	1.5×10^{-2}
2.0×10^{-1}	1.1×10^{-2}	2.0×10^{-1}	1.1×10^{-2}	2.0×10^{-1}	1.1×10^{-2}
4.6×10^{-2}	6.6×10^{-3}	4.6×10^{-2}	6.6×10^{-3}	4.6×10^{-2}	6.6×10^{-3}

(b) RMS and normalized RMS for auto-translated OpenCL on non-GPU platforms.

Table 4: Accuracy of GEM's surface potential calculation is demonstrated via root mean square error (RMS) analysis.

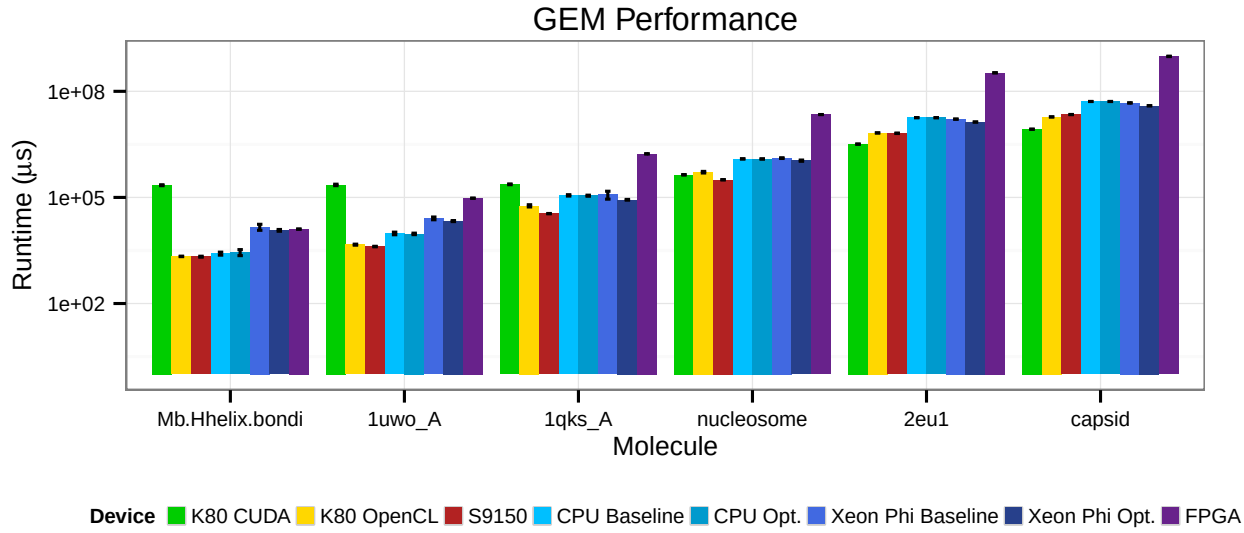

Figure 7: A comparison of the GEM runtimes.

Fig. 7 shows the run times of the full solver for a range of accelerator devices. Included in the run times are the device allocations, host/device data transfers and one or more kernel invocations, as necessary, to accumulate the charge from all atoms to all vertices. As with LDC, one-time runtime initializations, including OpenCL JIT compilations, are performed before the timed region.

Examining the performance of a computationally dense N-body kernel across five orders of magnitude of simulation complexity (in terms of the number of atom × vertex calculations) across a range of computational platforms yields some interesting performance insights. Despite accounting for the high one-time initialization cost of the CUDA runtime outside the timed region (approximately five seconds), an approximately 200ms fixed cost is still present, demonstrated by the lower-bound CUDA run time for the three smallest molecules. This fixed cost is not associated with either the kernel invocation or data transfers, as both are enclosed within their own timing sub-region, implying the cost is paid during the `cudaAlloc` calls, regardless of the size of the device allocation.

The OpenCL implementations for the CPU and GPU perform the best at the smallest problem size (Mb.HHelix), where the performance is dominated by runtime effects, and the Xeon Phi and Altera FPGA run times have moderate fixed costs, but not as severe as CUDA. The CPU begins to lag behind the GPUs by the second-smallest test case (1uwo_A), and the FPGA begins to lag further behind the Xeon Phi. By the third test case (1qks_A), the FPGA is orders of magnitude slower than the other platforms and persists through the larger tests. A deep analysis of how to improve the performance of GPU-optimized codes on FPGA remains as future work. In addition, with the third test (1qks_A) and fourth test (nucleosome), the AMD GPU achieves the best performance despite the significantly more expensive `__shfl` emulation that it is performing versus the original CUDA's `__shfl` intrinsic. At this point, the advantage of the CPU's more efficient runtime system over the Xeon Phi's tapers off, and the two devices remain close to each other in performance through the larger tests.

The largest two tests (i.e., 2eu1 and capsid biomolecules) are sufficiently large to amortize the fixed overheads, and thus, provide additional insight into the performance portability of auto-translated codes. The advantage of the `__shfl` atom exchange is apparent as the original CUDA implementation achieves the best performance on these two large biomolecules — 2eu1 and capsid. The OpenCL implementation on the AMD and NVIDIA GPUs delivers remarkably consistent performance across vendors. At scale the Xeon Phi gains a slight performance edge over the CPU.

While running the GEM molecular modeling code on Xeon Phi is known to be capable of matching and even exceeding the performance of running on the NVIDIA GPU [18], our auto-translated OpenCL code actually performs worse on the Xeon Phi than on the NVIDIA GPU. So, we investigated the Xeon Phi performance using Intel's VTune advanced hotspot analysis, as done for LDC. In contrast to LDC, however, we could not identify a clear cause for the poor performance. The GEM kernel is compute-bound, and the profiling data shows that the performance is fairly distributed across both mathematical operations and memory transactions. Further, the cycles per instruction (CPI) is less than the four that VTune indicates as a hot function.

By manually applying two optimizations to the Ope to improve performancenCL implementation for Xeon Phi — (1) replacing division by a square root with a reciprocal square-root operation and (2) preloading the first atom to be processed by all threads before the hot loop, we improved the performance by and additional 1.2× over the baseline-auto-translated CUDA-to-OpenCL code on Xeon Phi and caused no negative effects to the efficiency of the CPU OpenCL execution.

5 RELATED WORK

Platform portability has served as a driver for programming language and runtime system design. For accelerators, we see this same desire for an approach that provides "write once, run *efficiently* anywhere" behavior. As CUDA was the first GPU programming model to gain significant traction, many attempts to deliver platform portability have sought to bridge between CUDA or its PTX intermediate representation (IR) and other platforms. These efforts generally take one or more of the following approaches:

- A unified runtime layer implemented *on top of* CUDA and one or more other models that support other devices
- Automatic source-level translation of kernel and/or host code
- Automatic conversion of PTX IR to another device's IR
- Runtime compatibility or wrapper layers *between* CUDA and one or more other models

Here we leveraged CU2CL [12], a compiler-based CUDA-to-OpenCL auto-translator for both kernel and host code, that uses a small amount of on-the-fly generated runtime compatability code to construct functionally-equivalent analogs to CUDA functions. Closely related is the work of Kim et al. [16], which provides hand-written bi-directional runtime wrapper layers between CUDA and OpenCL host code and a kernel translator; this work is distinguished from CU2CL by providing an OpenCL-to-CUDA compatibility layer but not aiming to provide a static translation of the host runtime API and providing a compatibility layer instead. Thus, the two projects have complementary goals: Kim et al. needs an expansive third-party compatibility layer to *execute* CUDA code on OpenCL platforms while we take a static translation approach to *port* a CUDA application to pure OpenCL for continued development.

MCUDA [24] provides a compiler-based CUDA-to-Pthreads translation of both host and kernel code but uses a runtime layer to map the CUDA host API to standard libc calls. Other related work falls in the category of PTX-to-X translation, where X is some other intermediate representation, e.g., Ocelot [8] and Caracal [9] provide such functionality for x86 and AMD Compute Abstraction Layer (CAL) [2], respectively, but rely on a third-party implementation of the CUDA runtime API to manage and launch the translated PTX.

6 CONCLUSION

Automatic and portable translation from CUDA to OpenCL is a practical tool for enabling CUDA applications to run on other accelerators, including AMD and NVIDIA GPUs, CPUs, Xeon Phi, and even FPGAs. We demonstrate that CU2CL can achieve portability and maintain or even improve application performance on the same or different devices via three significant scientific applications: (1) lid-driven cavity (LDC) for computational fluid dynamics, (2) *Fen Zi* for molecular dynamics, and (3) *GEM* for molecular modeling. Furthermore, even without re-optimization, we show that CUDA-optimized codes can run on AMD GPUs via static translation and achieve up to a 3.1× speedup over the original CUDA-optimized code.

ACKNOWLEDGEMENTS

The authors wish to thank Salvatore Pezzino for his post-translation debugging and validation of the OpenCL variant of Fen Zi.

REFERENCES

[1] CU2CL: A Prototype CUDA-to-OpenCL Source-to-Source Translator. https://github.com/vtsynergy/CU2CL.
[2] Advanced Micro Devices (AMD). *AMD Compute Abstraction Layer (CAL)*, December 2010.
[3] Anandakrishnan, R., Scogland, T. R., Fenley, A. T., Gordon, J. C., chun Feng, W., and Onufriev, A. V. Accelerating electrostatic surface potential calculation with multi-scale approximation on graphics processing units. *Journal of Molecular Graphics and Modelling 28*, 8 (2010), 904 – 910.
[4] Cabo-Bilbao, A., Spinelli, S., Sot, B., Agirre, J., Mechaly, A. E., Muga, A., and Guerin, D. M. Crystal structure of the temperature-sensitive and allosteric-defective chaperonin groele461k. *Structural Biology 155*, 1047-8477 (Linking) (2006), 482–492.
[5] Daga, M., Scogland, T., and c. Feng, W. Architecture-aware mapping and optimization on a 1600-core gpu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on* (Dec 2011), pp. 316–323.
[6] Davey, C. A., Sargent, D. F., Luger, K., Maeder, A. W., and Richmond, T. J. Solvent mediated interactions in the structure of the nucleosome core particle at 1.9 a resolution. *Molecular Biology 319*, 0022-2836 (Linking) (2002), 1097–1113.
[7] del Mundo, C., and Feng, W.-c. Towards a performance-portable fft library for heterogeneous computing. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (New York, NY, USA, 2014), CF '14, ACM, pp. 11:1–11:10.
[8] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *19th International Conference on Parallel Architectures and Compilation Techniques* (2010), pp. 353–364.
[9] Domínguez, R., Schaa, D., and Kaeli, D. Caracal: Dynamic Translation of Runtime Environments for GPUs. In *4th Workshop on General Purpose Processing on Graphics Processing Units* (2011), pp. 5:1–5:7.
[10] Fenley, A. T., Gordon, J. C., and Onufriev, A. An analytical approach to computing biomolecular electrostatic potential. i. derivation and analysis. *The Journal of Chemical Physics 129*, 7 (2008).
[11] Fulop, V., Moir, J. W. B., Ferguson, S. J., and Hajdu, J. The anatomy of a bifunctional enzyme: structural basis for reduction of oxygen to water and synthesis of nitric oxide by cytochrome cd1. *Cell 81*, 0092-8674 (Linking) (1995).

[12] Gardner, M., Sathre, P., chun Feng, W., and Martinez, G. Characterizing the challenges and evaluating the efficacy of a cuda-to-opencl translator. *Parallel Computing 39*, 12 (2013), 769 – 786. Programming models, systems software and tools for High-End Computing.

[13] Gordon, J. C., Fenley, A. T., and Onufriev, A. An analytical approach to computing biomolecular electrostatic potential. ii. validation and applications. *The Journal of Chemical Physics 129*, 7 (2008).

[14] Group, T. K. Khronos releases opencl 2.2 provisional specification with opencl c++ kernel language for parallel programming, April 18 2016.

[15] Huang, S., Xiao, S., and Feng, W. On the energy efficiency of graphics processing units for scientific computing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (May 2009), pp. 1–8.

[16] Kim, J., Dao, T. T., Jung, J., Joo, J., and Lee, J. Bridging opencl and cuda: A comparative analysis and translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 82:1–82:12.

[17] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Kobayashi, H. Evaluating performance and portability of opencl programs. In *The Fifth International Workshop on Automatic Performance Tuning* (June 2010).

[18] Krommydas, K., Scogland, T. R. W., and Feng, W.-C. On the programmability and performance of heterogeneous platforms. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on* (Dec 2013), pp. 224–231.

[19] Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H. Top 500 supercomputers, 2018.

[20] Pickering, B. P., Jackson, C. W., Scogland, T. R., Feng, W.-C., and Roy, C. J. Directive-based {GPU} programming for computational fluid dynamics. *Computers & Fluids 114* (2015), 242 – 253.

[21] SE, P. Structure and refinement of oxymyoglobin at 1.6 a resolution. *Molecular Biology 142*, 0022-2836 (Linking) (1980), 531–554.

[22] Sochacki, B., Ed. *The OpenCL C++ Specification: Version 1.0, Revision 22.* Khronos OpenCL Working Group, 2016. https://www.khronos.org/registry/cl/specs/opencl-2.2-cplusplus.pdf.

[23] SP, S., and GS, S. A novel calcium-sensitive switch revealed by the structure of human s100b in the calcium-bound form. *Structure 6*, 0969-2126 (Linking) (1998), 211–222.

[24] Stratton, J. A., Stone, S. S., and Hwu, W.-M. W. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30.

[25] Taufer, M., Ganesan, N., and Patel, S. Gpu-enabled macromolecular simulation: Challenges and opportunities. *Computing in Science Engineering 15*, 1 (Jan 2013), 56–65.

[26] V, C., and JE, J. The structure of tobacco ringspot virus: a link in the evolution of icosahedral capsids in the picornavirus superfamily. *Structure 6*, 0969-2126 (Linking) (1998), 157–171.