25th IEEE Int'l Conference on Application-Specific Systems, Architectures and Processors (ASAP), Zurich, Switzerland, June 18-20, 2014
Best Paper Finalist

On the Characterization of OpenCL Dwarfs on Fixed and Reconfigurable Platforms

Konstantinos Krommydas*, Wu-chun Feng*, Muhsen Owaida†, Christos D. Antonopoulos†, Nikolaos Bellas†
*Department of Computer Science, Virginia Tech
†Department of Electrical and Computer Engineering, University of Thessaly
E-mails: {kokrommy, wfeng}@vt.edu, {mowaida, cda, nbellas}@uth.gr

***Abstract*—The proliferation of heterogeneous computing platforms presents the parallel computing community with new challenges. One such challenge entails evaluating the efficacy of such parallel architectures and identifying the architectural innovations that ultimately benefit applications. To address this challenge, we need benchmarks that capture the execution patterns (i.e., dwarfs or motifs) of applications, both present and future, in order to guide future hardware design. Furthermore, we desire a common programming model for the benchmarks that facilitates code portability across a wide variety of different processors (e.g., CPU, APU, GPU, FPGA, DSP) and computing environments (e.g., embedded, mobile, desktop, server).**

As such, we present the latest release of OpenDwarfs, a benchmark suite that currently realizes the Berkeley dwarfs in OpenCL, a vendor-agnostic and open-standard computing language for parallel computing. Using OpenDwarfs, we characterize a diverse set of fixed and reconfigurable parallel platforms: multi-core CPUs, discrete and integrated GPUs, Intel Xeon Phi co-processor, as well as a FPGA. We describe the computation and communication patterns exposed by a representative set of dwarfs, obtain relevant profiling data and execution information, and draw conclusions that highlight the complex interplay between dwarfs' patterns and the underlying hardware architecture of modern parallel platforms.

Keywords*—*OpenDwarfs; benchmarking; evaluation; dwarfs; performance characterization; CPU; FPGA; GPU; OpenCL

I. Introduction

Over the span of the last decade, the computing world has borne witness to a parallel computing revolution, which delivered parallel computing to the masses while doing so at low cost. The programmer has been presented with a myriad of new computing platforms promising ever-increasing performance. Programming these platforms entails familiarizing oneself with a wide gamut of programming environments, along with optimization strategies strongly tied to the underlying architecture. The aforementioned realizations present the parallel computing community with two challenging problems: (a) the need of a common means of programming, and (b) the need of a common means of evaluating this diverse set of parallel architectures.

The former problem was effectively solved through a concerted industry effort that led to a new parallel programming model, i.e., OpenCL. Efforts, like SOpenCL [1] and Altera OpenCL [2] enable transforming OpenCL kernels to equivalent synthesizable hardware descriptions, thus facilitating exploitation of FPGAs as hardware accelerators, while obviating the overhead of additional development cost and expertise.

The latter problem cannot be sufficiently addressed by the existing benchmark suites. Such benchmarks suites (e.g., SPEC CPU [3], PARSEC [4]) are often written in a language tied to a particular architecture and porting the benchmarks to another platform would typically mandate re-writing them using the programming model suited for the platform under consideration. The additional caveat in simply re-casting these benchmarks as OpenCL implementations is that existing benchmark suites represent collections of overly specific applications that do not address the question of what the best way of expressing a parallel computation is. This impedes innovations in hardware design, which will come as a quid pro quo, only when software idiosyncrasies are taken into account at design and evaluation stages. This is not going to happen unless software requirements are abstracted in a higher level and represented by a set of more meaningful benchmarks. To address all these issues, we proposed OpenDwarfs [5], a benchmark suite for heterogeneous computing in OpenCL, in which applications are selected based on the computation and communication patterns defined by Berkeley's Dwarfs [6].

Our contributions in this paper are two-fold:

(a) We present the latest OpenDwarfs release, in which we attempt to rectify prior release's shortcomings. We propose and implement all necessary changes towards a comprehensive benchmark suite that adheres both to the dwarfs' concept and established benchmark creation guidelines.
(b) We verify functional portability and characterize OpenDwarfs' performance on multi-core CPUs, discrete and integrated GPUs, the Intel Xeon Phi co-processor and even FPGAs, and relate our observations to the underlying computation and communication pattern of each dwarf.

The rest of the paper is organized as follows: in Section II we discuss related work. Section III presents our latest contributions to OpenDwarfs. In Section IV we offer an overview of FPGA architectures and the SOpenCL tool. Section V outlines our experimental setup, followed by results in Section VI. Section VII concludes the paper and discusses future work.

II. Related Work

HPC engineering and research have highlighted the importance of developing benchmarks that capture high-level computation and communication patterns. In [7] the authors emphasize the need for benchmarks to be related to scientific *paradigms*, where a paradigm defines what the important problems in a scientific domain are and what the set of accepted solutions is. This notion of paradigm parallels that

TABLE I: Dwarf Instantiations in OpenDwarfs

Dwarf	Dwarf Instantiation
Dense Linear Algebra	LUD (LU Decomposition)
Sparse Matrix-Vector Matrix Multiplication	CSR (Compressed Sparse-Row Vector Multiplication)
Graph Traversal	BFS (Breadth-First Search)
Spectral Methods	FFT (Fast Fourier Transform)
N-body Methods	GEM (Electrostatic Surface Potential Calculation)
Structured Grid	SRAD (Speckle Reducing Anisotropic Diffusion)
Unstructured Grid	CFD (Computational Fluid Dynamics)
Combinational Logic	CRC (Cyclic Redundancy Check)
Dynamic Programming	NW (Needleman-Wunsch)
Backtrack & Branch and Bound	NQ (N-Queens Solver)
Finite State Machine	TDM (Temporal Data Mining)
Graphical Models	HMM (Hidden Markov Model)
MapReduce	StreamMR

of the *computational dwarf*. A dwarf is an algorithmic method that encapsulates a specific computation and communication pattern. The seven original dwarfs, attributed to P. Colella's unpublished work, became known as *Berkeley's dwarfs*, after Asanovic et al. [6] formalized the dwarf concept and complemented the original set of dwarfs with six more. Based in part on the dwarfs, Keutzer et al. later attempted to define a pattern language for parallel programming [8].

The combination of the aforementioned works sets a concrete theoretical basis for benchmark suites. Following this path and based on the very same nature of the dwarfs and the global acceptance of OpenCL, our work on extending OpenDwarfs attempts to present an all-encompassing benchmark suite for heterogeneous computing. Such a benchmark suite, whose application selection delineates modern parallel application requirements, can constitute the basis for comparing and guiding hardware and architectural design. On a parallel path with OpenDwarfs, which was based on OpenCL from the onset, many existing benchmark suites were re-implemented in OpenCL and new ones were released (e.g., Rodinia [9], SHOC [10], Parboil [11]). Most of them were originally developed as GPU benchmarks and as such still carry optimizations that favor GPU platforms. This violates the *portability* requirement for benchmarks that mandates a lack of bias for one platform over another [6], [7] and prevents drawing broader conclusions with respect to hardware innovations. We attempt to address the above issues with our efforts in extending OpenDwarfs.

On the practical side of matters, benchmark suites are used for characterizing architectures. Both [9] and [10] discuss architectural differences between CPUs and GPUs on a higher level. Although not based on OpenCL kernels, a more detailed discussion on architectural features' implications with respect to algorithms and insight on future architectural design requirements is given in [12]. In this work, we complement prior research by characterizing OpenDwarfs on a diverse set of modern parallel architectures, including CPUs, APUs, discrete GPUs, the Intel Xeon Phi co-processor, as well as on FPGAs.

III. OpenDwarfs Benchmark Suite

OpenDwarfs is a benchmark suite that comprises 13 of the dwarfs, as defined in [6]. The dwarfs and their corresponding instantiations (i.e., applications) are shown in Table I. This OpenDwarfs release provides full coverage of the dwarfs, including more stable implementations of the *Finite State Machine* and *Backtrack & Branch and Bound* dwarfs. CSR (*Sparse Linear Algebra* dwarf) and CRC (*Combinational Logic* dwarf) have been extended to allow for a wider range of options, including running with varying work-group sizes or running the main kernel multiple times. We plan to propagate these changes to the rest of the dwarfs, as they can uncover potential performance issues for each of the dwarfs on devices of different capabilities.

One of the most important changes in this implementation of OpenDwarfs is related to the *uniformity* of optimization level across all dwarfs. More precisely, none of the dwarfs contains optimizations that would make a specific architecture more favorable than another. Use of shared memory, for instance, in many of the dwarfs in the previous OpenDwarfs release favored GPU architectures. Such favoritism limits the scope of a benchmark suite, as we discuss in Section II.

In order to enhance code uniformity, readability and usability for our benchmark suite, we have augmented the OpenDwarfs library of common functions. For example, we introduce more uniform error checking, while a set of common options can be used to select and initialize the appropriate OpenCL device at run-time. FPGA support for Altera FPGAs is offered, but currently limited to two of the dwarfs, due to lack of complete support of the OpenCL standard by the Altera OpenCL SDK, which requires certain alterations to the code for successful compilation and full FPGA compatibility [13]. We plan to provide full coverage in upcoming releases, but for completeness in the context of this work we use SOpenCL that enables full Xilinx FPGA OpenCL support.

IV. FPGA Technology and SOpenCL

FPGAs (*field-programmable gate arrays*) are configured post-fabrication through configuration bits that specify the functionality of the configurable high-density arrays of uncommitted logic blocks and the routing channels between them. They offer the highest degree of flexibility in tailoring the architecture to match the application and avoid the traditional ISA-based von Neumann architecture followed by CPUs and GPUs. FPGAs are traditionally programmed using Hardware Description Languages (VHDL or Verilog), a time-consuming and laborious task that requires deep knowledge of low-level hardware details.

We use the SOpenCL tool [1] to automatically generate hardware accelerators for the OpenDwarf kernels, thus dramatically minimizing development time and increasing productivity. The SOpenCL front end is a source-to-source compiler that adjusts parallelism granularity of the OpenCL kernel to better match the hardware capabilities of the FPGA. The output of this stage is semantically equivalent C code at the work-group granularity. SOpenCL back-end flow is based on the LLVM compiler infrastructure [14], supports *bitwidth optimization*, *predication*, and *swing modulo scheduling* (SMS) and generates the synthesizable Verilog of the accelerator.

V. Experimental Setup

This section presents our experimental setup. First, we present the software setup and methodology used for collecting the results and discuss the hardware used in our experiments.

TABLE II: Configuration of the Target Fixed Architectures

Model	AMD Opteron 6272	AMD Llano A8-3850	AMD Radeon HD 6550D	AMD A10-5800K	AMD Radeon HD 7660D	AMD Radeon HD 7970	Intel Xeon Phi P1750
Type	CPU	CPU*	Integr. GPU*	CPU*	Integr. GPU*	Discrete GPU	Co-processor
Frequency	2.1 GHz	2.9 GHz	600 MHz	3.8 GHz	800 MHz	925 MHz	1.09 GHz
Cores	16	4	5†	4	6†	32†	61
Threads/core	1	1	5	1	4	4	4
L1/L2/L3 Cache (KB)	16/2048/ 8192‡	64/1024/- (per core)	8/128/- (L1 per CU)	64/2048/- (per 2 cores)	8/128/- (L1 per CU)	16/768/- (L1 per CU)	32/512/- (per core)
SIMD (SP)	4-way	4-way	16-way	8-way	16-way	16-way	16-way
Process	32nm	32nm	32nm	32nm	32nm	32nm	22nm
TDP	115W	100W*	100W*	100W*	100W*	210W	300W
GFLOPS (SP)	134.4	46.4	480	121.6	614.4	3790	2092.8

† Compute Units (CU) ‡ L1: 16KBx16 data shared, L2: 2MBx8 shared, L3: 8MBx2 shared * CPU and GPU fused on the same die, total TDP

TABLE III: OpenDwarfs Benchmark Test Parameters

Benchmark	Problem Size and Index Space
GEM	Input file & parameters: nucleosome 80 1 0.
NW	Two protein sequences of 4096 letters each.
SRAD	2048x2048 FP matrix, 128 iterations.
BFS	Graph: 248,730 nodes and 893,003 edges.

A. Software and experimental methodology

For benchmarking our target architectures we use the latest release of OpenDwarfs (available for download at *https://github.com/opendwarfs/OpenDwarfs*). The CPU/GPU/APU software environment consists of 64-bit Debian Linux 7.0 with kernel version 2.6.37, GCC 4.7.2 and AMD APP SDK 2.8. AMD GPU/APU drivers are AMD Catalyst 13.1. Intel Xeon Phi is hosted on a CentOS 6.3 environment with the Intel SDK for OpenCL applications XE 2013. For profiling we use AMD CodeXL 1.3 and Intel Vtune Amplifier XE 2013 for the CPU/GPU/APU and Intel Xeon Phi, respectively. In Table III we provide details about the subset of dwarf applications used and their input datasets and/or parameters. Kernel execution time and data transfer times are accounted for and measured by use of the corresponding OpenDwarfs timing infrastructure. In turn, the aforementioned infrastructure lies on the OpenCL events to provide accurate timing to a very fine granularity.

B. Hardware

In order to capture a wide range of parallel architectures, we pick a set of representative device types: a high-end multi-core CPU (AMD Opteron 6272) and a high-performance discrete GPU (AMD Radeon HD 7970). An integrated GPU (AMD Radeon HD 6550D) and a low-powered low-end CPU (A8-3850), both part of a heterogeneous Llano APU system (i.e., CPU and GPU fused on the same die), as well as a newer generation APU system (Trinity) comprising an A10-5800K and an AMD Radeon HD 7660D integrated GPU. Finally, an Intel Xeon Phi co-processor. Details for each of the aforementioned architectures are given in Table II. To evaluate OpenDwarfs on FPGAs, we use the Xilinx Virtex-6 LX760 FPGA on a PCIe v2.1 board, which consumes approximately 50 W and contains 118560 logic slices. Each slice includes 4 LUTs and 8 flip-flops. FPGA clock frequency ranges from 150 to 200 MHz for all designs.

VI. Results

Here we present our results of running a representative subset of the dwarfs on a wide array of parallel architectures.

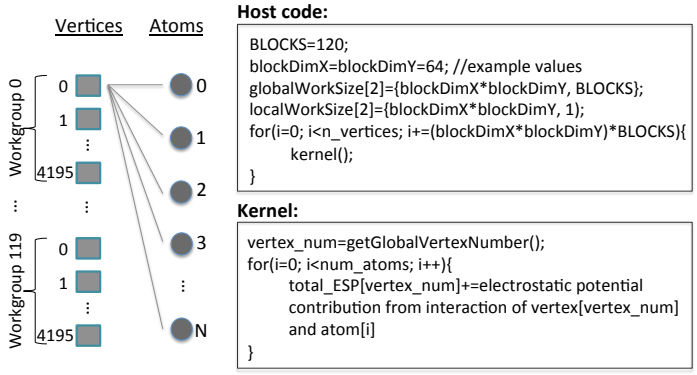


Fig. 1: GEM

After we verify functional portability across all platforms, including the FPGA, we characterize the dwarfs and illustrate their utility in guiding architectural innovation, which is one of the main premises of the OpenDwarfs benchmark suite.

A. N-body Methods: GEM

The n-body class of algorithms refers to those algorithms that are characterized by all-to-all computations within a set of particles (bodies). In the case of GEM, our n-body application, the electrostatic surface potential of a biomolecule is calculated as the sum of charges contributed by all atoms in the biomolecule due to their interaction with a specific surface vertex (two sets of bodies). In Figure 1 we illustrate the computation pattern of GEM and present the pseudocode running on the OpenCL host and device. Each work-item accumulates the potential at a single vertex due to every atom in the biomolecule. A number of work-groups (*BLOCKS=120* in our example) each having *blockDimX*blockDimY* work-items (*4096* in our example) is launched, until all vertices' potential has been calculated.

GEM's computation pattern is regular, in that the same amount of computation is performed by each work-item in a work-group and no dependencies hinder computation continuity. Total execution time is mainly dependent on the maximum computation throughput. Computation itself is characterized by FP arithmetic, including (typically expensive) division and square root operations that constitute one of the main bottlenecks. Special hardware can provide low latency alternatives of these operations, albeit at the cost of minor accuracy loss that may or may not be acceptable for certain types of

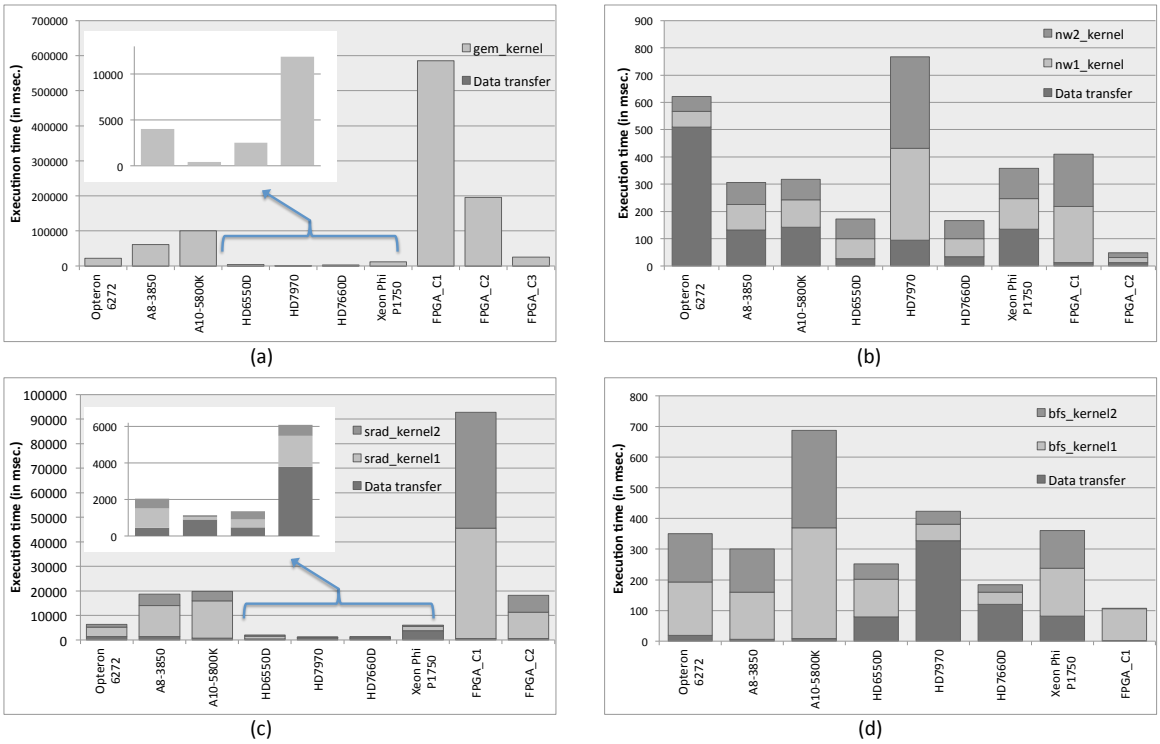


Fig. 2: Results: (a) GEM, (b) NW, (c) SRAD, (d) BFS

applications. Such fast math implementations are featured in many architectures and typically utilize look-up tables for fast calculations.

With respect to data accesses, atom data is accessed in a serial pattern, simultaneously by all work-items. This facilitates efficient utilization of cache memories available in each architecture. Figure 2 and Table II can assist in pinpointing which architectural features are important for satisfactory GEM performance: good FP performance and sufficient first level cache. With respect to the former, Opteron 6272 and A10-5800K CPUs reach about 130 GFLOPS and A8-3850 falls behind by a factor of 2.9, as defined by their number of cores, SIMD capability and core frequency. However, the cache hierarchy between the three CPU architectures is fundamentally different. Opteron 6272 has 16K of L1 cache per core, which is *shared* among all 16 cores. Given the computation and communication pattern of n-body dwarfs, such types of caches may be an efficient choice. Cache miss rates at this level (L1), are also indicative of the fact: A8-3850 with 64KB of dedicated L1 cache per core is characterized by a 0.55% L1 cache miss rate, with Opteron 6272 at 10.2% and A10-5800K a higher 24.25%. Those data accesses that result in L1 cache misses are mostly served by L2 cache and rarely require expensive RAM memory accesses. Measured L2 cache miss rates are 4.5%, 0.18% and 0%, respectively, reflecting the L2 cache capability of the respective platforms (Table II). Of course, the absolute number of accesses to L2 cache, depend on the previous level's cache misses, so a smaller percentage on a platform, tells only part of the story if we plan to compare different platforms to each other. In cases where data accesses follow a predictable pattern, like in GEM, specialized hardware can predict what data is going to be needed and fetch it ahead of time. Such *hardware prefetch* units are available - and of advanced maturity - in multi-core CPUs. This proactive loading of data can take place between the main memory and last level cache (LLC) or between different cache levels. In all three CPU platforms, a large number of prefetch instructions is emitted, as seen through profiling the appropriate counter, which, together with the regular data access patterns, verify the overall low L1 cache miss rates mentioned earlier.

Xeon Phi's execution is characterized by high vectorization intensity (12.84, the ideal being 16), which results from regular data access patterns and implies efficient auto-vectorization on behalf of the Intel OpenCL compiler and its *implicit vectorization module*. However, profiling reveals that the estimated latency impact is high indicating that the majority of L1 misses result in misses in L2 cache, too. This signifies the need for optimizations such as data reorganization and blocking for L2 cache, or the introduction of a more advanced hardware prefetch unit in future Xeon Phi editions - currently there is lack of automatic (i.e., hardware) prefetching to L1 cache (only main memory to L2 cache prefetching is supported). Further enhancement of the ring interconnect that allows efficient sharing of the dedicated (per core) L2 cache contents across cores would also assist in attaining better performance for the n-body dwarf. While Xeon Phi, lying between the multi-core CPU and many-core GPU paradigms, achieves good overall performance for this - unoptimized, architecture agnostic - code implementation, it falls behind its theoretical maximum performance of nearly 2 TFLOPS.

With respect to GPU performance, raw FP performance is one of the deciding factors, as well. As a result HD 7970 performs the best and is characterized by the best occupancy

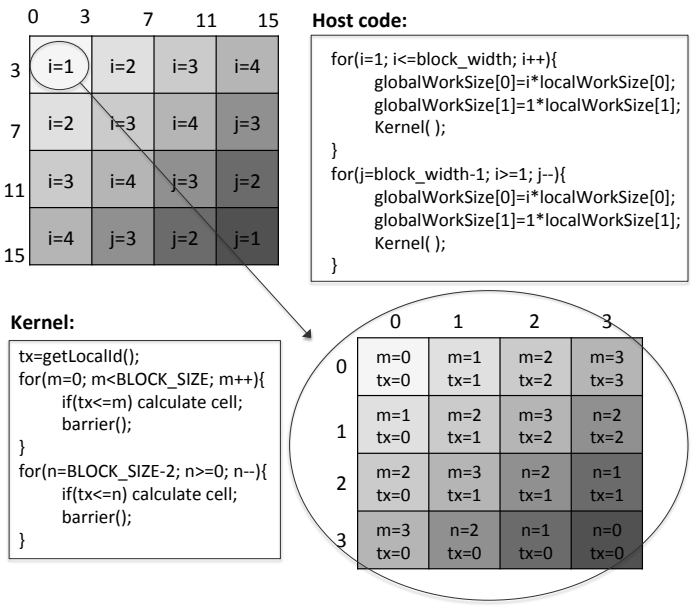


Fig. 3: Needleman-Wunsch

(70%), compared to 57.14% and 37.5% for HD 7660D and HD 6550D, respectively. In all three cases, cache hit rates are over 97% (reaching 99.96% for HD 7970, corroborating that our conclusions for the CPU cache architectures hold for GPUs, too, for this class of applications (i.e, n-body dwarf). Correspondingly, the measured percentage of memory unit stalls is held at low levels. In fact, the memory unit is kept busy for over 76% of the time for all three GPU architectures, including all extra fetches and writes and taking any cache or memory effects into account.

Although FPGAs are not made for FP performance, SOpenCL produces accelerators whose performance lies between that of CPUs and GPUs. SOpenCL instantiates modules for single-precision FP operations, such as division and square root. Partially unrolling the outer loop executed by each thread four times results in nearly 4-fold speedup (FPGA_C2) compared to the base accelerator configuration (FPGA_C1). Multiple accelerators can be instantiated and process in parallel different vertices on the grid, thus providing even higher speedup (FPGA_C3).

B. Dynamic Programming: Needleman-Wunsch (NW)

Dynamic programming is a programming method in which a complex problem is solved by decomposition into smaller subproblems. Combining the solutions to the subproblems provides the solution to the original problem. Our dynamic programming dwarf, Needleman-Wunsch, performs protein sequence alignment, i.e., attempts to identify the similarity level between two given strings of aminoacids. Figure 3 illustrates its computation pattern and two levels of parallelism. Each element of the 2D matrix depends on the values of its west, north and north-west neighbors. This set of dependencies limits available parallelism and enforces a wave-front computation pattern. On the first level blocks of computation (i.e., OpenCL work-groups) are launched across the anti-diagonal and on the second level, each of the work-group's work-items works on cells on each anti-diagonal. Available parallelism at each stage is variable, starting with a single work-group, increasing as we reach the main anti-diagonal and decreasing again as we reach the bottom right. Parallelism varies within each work-group in a similar way, as shown in the respective figure, where a variable number of work-items work independently in parallel at each anti-diagonal's level. Needleman-Wunsch algorithm imposes significant synchronization overhead (repetitive barrier invocation within the kernel) and requires modest integer performance. Computations for each 2D matrix cell entail calculating an alignment score that depends on the three neighboring entries (west, north, northwest) and a max operation (i.e., nested if statements).

In algorithms like NW that are characterized by inter- and intra-work-group dependencies there are two big considerations. First, the overhead for repetitively launching a kernel (corresponding to inter-work-group synchronization), and second, the cost of the intra- work-group synchronization via *barrier()* or any other synchronization primitives. Introduction of system-wide (hardware) barriers would help to solve the former of the problems, while optimization of already existing intra-work-group synchronization primitives would be beneficial for this kind of applications for the latter case.

Memory accesses follow the same pattern as computation, i.e., for each element the west, north and northwest elements are loaded from the reference matrix. For each anti-diagonal *m* within a work-group (Figure 3) the updated data from anti-diagonal *m-1* is used.

As we can observe, GPUs do not perform considerably better than the CPUs. In fact, Opteron 6272 surpasses all GPUs (and even Xeon Phi), when we only take kernel execution time into account. What needs to be emphasized in the case of algorithms, such as NW, is the variability in the characteristics of each kernel iteration. In Figure 6a we observe such variability for metrics like the percentage of the time the ALU is busy, the cache hit rate, the fetch unit is busy or stalled, on the HD 7660D. Similar behavior is observed in the case of HD 6550D. Most of these metrics can be observed to be a function of the number of active wavefronts in every kernel launch. For instance, cache hit follows an inverse-U-shaped curve, as do most of the aforementioned metrics. In both cases, occupancy is below 40% (25% for HD 6550D) and ALU packing efficiency barely reaches 50%, which indicates a mediocre job on behalf of the shader compiler in packing scalar and vector instructions as VLIW instructions of the Llano and Trinity integrated GPUs (i.e., HD 6550D and HD 7660D).

As expected, the FPGA performs the best when it comes to integer code, in which case, its performance lies closer to GPUs than to CPUs. Multiple accelerators (5 pairs) and fully unrolling the innermost loop deliver higher performance (FPGA_C2) than a single pair (FPGA_C1) and render the FPGA implementation the fastest choice for the dynamic programming dwarf. In the FPGA implementation of NW, the data fetches' pattern favors decoupling of the compute path from the *data fetch & fetch address generation unit*, as well as from the *data store & store address generation unit*. This allows aggressive data prefetching in buffers ahead of time of the actual data requests.

Host code:

```
Loop for iter number of iterations{
        calculate statistics for the region of interest
        blockX=columns/BLOCK_SIZE;
        blockY=rows/BLOCK_SIZE;
        localWorkSize[2]={BLOCK_SIZE, BLOCK_SIZE};
        globalWorkSize[2]={blockX*localWorkSize[0],
                           blockY*localWorkSize[1]};
        kernel1();
        kernel2();
}
```

Kernel1:

```
(Each work-item (i,j) works on a 2D table element)
dN[i][j]=J[north][j]-J[i][j];
dS[i][j]=J[south][j]-J[i][j];
dW[i][j]=J[i][west]-J[i][j];
dE[i][j]=J[i][east]-J[i][j];
Calculate various parameters based above
values & initial J[i][[j] value;
Using the above value, calculate diffusion
coefficient c[i][j];
```

Kernel2:

```
(Each work-item (i,j) works on a 2D table element)
cN=c[i][j];
cS=c[north][j];
cW=c[i][j];
cE=c[i][east];
D=cN*dN[i][j]+cS*dS[i][j]+cW*dW[i][j]+cE*dE[i][j];
J[i][j]=J[i][j]+0.25*lambda*D;
```

Fig. 4: SRAD

C. Structured Grids: Speckle Reducing Anisotropic Diffusion (SRAD)

Structured grids refers to those algorithms in which computation proceeds as a series of grid update steps. It constitutes a separate class of algorithms from unstructured grids, in that the data is arranged in a regular grid of two or more dimensions (typically 2D or 3D). SRAD is a structured grids application that attempts to eliminate speckles (i.e., locally correlated noise) from images, following a partial differential equation approach. Figure 4 presents a high-level overview of the SRAD algorithm, without getting into the specific details (parameters, etc.) of the method. Performance is determined by FP compute power. The computational pattern is characterized by a mix of FP calculations including divisions, additions and multiplications. Many of the computations in both SRAD kernels are in the form: $x = a * b + c * d + e * f + g * e$. These computations can easily be transformed by the compiler to multiply-and-add operations. In such cases, special *fused multiply-and-add* units can offer a faster alternative to the typical series of separate multiplication and addition. While such units are already existent, more instances can be beneficial for the structured grids dwarf.

A series of *if* statements (simple in *kernel1*, nested in *kernel2*) handles boundary conditions and different branches are taken by different work-items, potentially within the same work-group. Since boundaries constitute only a small part of the execution profile, especially for large datasets, these branches do not introduce significant divergence. In the case of CPU and Xeon Phi execution, branch misprediction rate never exceeded 1%, while on the GPUs *VALUUtilization* remained above 86% indicating a high number of active vector ALU threads in a wave and consequently minimal branch divergence and code serialization.

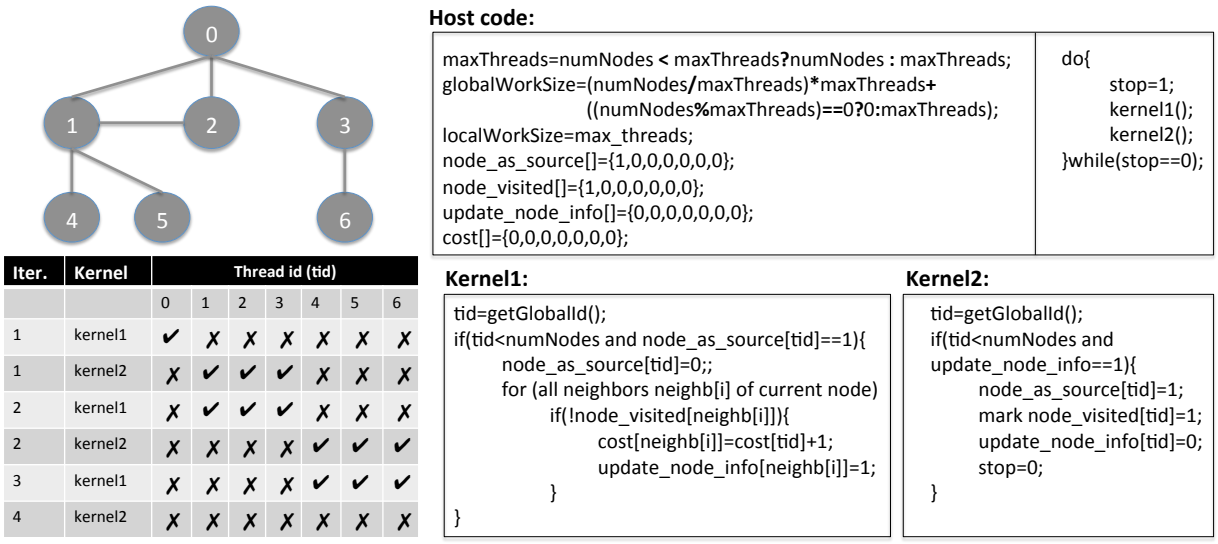
Following its computational pattern, memory access patterns in SRAD, as in all kinds of stencil computation, are localized and statically determined, an attribute that favors data parallelism. Although the data access pattern is a priori known, non-consecutive data accesses, prohibit ideal caching. As in the NW case, where data is accessed in a non-linear pattern, data locality is an issue here, too. Cache hit rates, especially for the GPUs, remain low (e.g., 33% for HD 7970). This leads to the memory unit being stalled for a large percentage of the execution time (e.g., 45% and 29% on average for HD 7970, for the two OpenCL kernels). Correspondingly, the vector and scalar ALU instruction units are busy for a small percentage of the total GPU execution time (about 21% and 5.6% for our example, on the two kernels on HD 7970). All this is highlighted by comparing performance across the three GPUs, and once more, indicates the need for advancements in the memory technology that would make fast, large caches more affordable for computer architects.

On the CPU and Xeon Phi side, large cache lines can afford to host more than one row of the 2D input data (depending on the input sequences' sizes). The huge L3 cache of Opteron 6272, along with its high core count, make it very efficient in executing this structured grid dwarf. In such algorithms, it is a balance between cache and compute power that distinguishes a good target architecture. Of course, depending on the input data set there are obvious trade-offs, as in the case of GPUs, which despite their poor cache performance are able to hide the latency by performing more computation simultaneously while waiting for the data to be available.

An FPGA implementation with a single pair of accelerators (one accelerator for each OpenCL kernel) offers performance worse even than that of the single-threaded Opteron 6272 execution (FPGA_C1). This is attributed mainly to the complex FP operations FPGAs are notoriously inefficient at. Multiple instances of these pairs of accelerators (five pairs in FPGA_C2) can process parts of the grid independently, bringing FPGA performance close to that of multicore CPUs. Different work-groups access separate portions of memory, hence multiple accelerators instances access different on-chip memories, keeping accelerators isolated and self-contained.

D. Graph Traversal: Breadth-First Search (BFS)

Graph traversal algorithms entail traversing a number of graph nodes and examining their characteristics. As a graph traversal application, we select a BFS implementation. BFS algorithms start from the root node and visit all the immediate neighbors. Subsequently, for each of these neighbors the corresponding (unvisited) neighbors are inspected, eventually leading to the traversal of the whole graph. BFS's computation pattern can be observed through a simple example (Figure 5), as well as by its host and device side pseudocode. The BFS algorithm's computation pattern is characterized by an *imbalanced* workload per kernel launch that depends on the



Iter.	Kernel	Thread id (tid)						
		0	1	2	3	4	5	6
1	kernel1	✔	✗	✗	✗	✗	✗	✗
1	kernel2	✗	✔	✔	✔	✗	✗	✗
2	kernel1	✗	✔	✔	✔	✗	✗	✗
2	kernel2	✗	✗	✗	✗	✔	✔	✔
3	kernel1	✗	✗	✗	✗	✔	✔	✔
4	kernel2	✗	✗	✗	✗	✗	✗	✗

Fig. 5: BFS

sum of the degrees deg(v_i) of the nodes at each level. For example (Figure 5), deg(v_0)=3, so only three work-items perform *actual* work in the first invocation of *kernel2*. Subsequently, *kernel1* has three working-items, as well. Second invocation of *kernel2* performs work on three nodes again (deg(v_1)+deg(v_2)+deg(v_3)=8, but nodes v_0, v_1, v_2 have already been visited, so effective deg(v_1)+deg(v_2)+deg(v_3)=3). Computation itself is negligible, being reduced to a simple addition with respect to each node's cost.

The way the algorithm works might lead to erroneous conclusions, if only occupancy and ALU utilization is taken into account, as in all three GPU cases it is over 95% and 88%, respectively (for both kernels). The problem lies in the fact that *not* all work-items perform useful work, and the fact that the kernels are characterized by reduced compute intensity (Figure 5). In such cases, up to a certain degree of problem size or for certain problem shapes, the number of compute units or frequency are not of paramount importance and high-end cards, like HD 7970 are about as fast as an integrated GPU (e.g., HD 7660D). The above is highlighted by the hardware performance counters that indicate poor ALU packing (e.g., 36.1% and 38.9% for the two BFS OpenCL kernels, on HD 7660D). Similarly, for HD 7970, the vector ALU is busy only for 5% (approximate value across kernel iterations) of the GPU execution time, even if the number of active vector ALU threads in the wave is high (*VALUUtilization*: 88.8%).

For similar reasons, CPU execution performance is capped on Opteron 6272, which performs only marginally better than A8-3850. It is interesting to see that A10-5800K and even Xeon Phi, with 8- and 16-way SIMD are characterized by lack of performance scalability. Why performance of A10-5800K is not *at least* similar to that of A8-3850 could not be pinpointed during profiling. However, in both A10-5800K and Xeon Phi cases, we found that the OpenCL compiler could not take advantage of the 256- and 512-bit wide vector unit, because of the very nature of graph traversal.

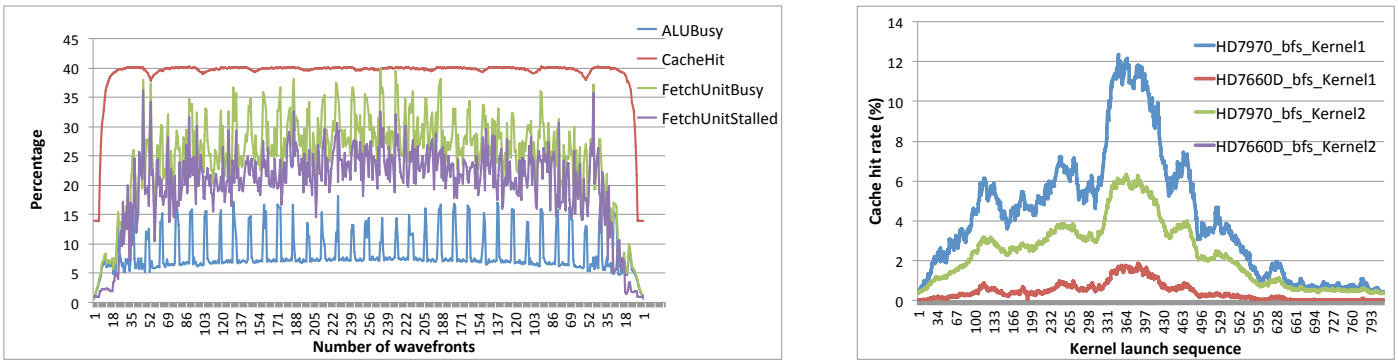
With respect to data accesses, BFS exhibits irregular access patterns. Each work-item accesses discontiguous memory locations, depending on the connectivity properties of the graph, i.e, how nodes of the current level being inspected are being connected to other nodes in the graph. Figure 5 is not only indicative of the resource utilization (work-items doing useful work), but of the inherent irregularity of memory accesses that depend on run-time assessed multiple levels of indirection, as well. Available caches' size define the cache hit rate, even in these cases, so HD 7970, which provides larger amounts of cache memory provides higher cache hit rates compared to the HD 7660D (varying for each kernel iteration, Figure 6b).

The FPGA implementation of BFS (FPGA_C1) is the fastest across all tested platforms. While *kernel1* is not as fast as in the fastest of our GPU platforms, minimal execution time for *kernel2* and data transfer time render it the ideal platform for graph traversal, despite the dynamic memory access pattern causing the input streaming unit to be merged with the data path, eliminating the possibility of aggressive data prefetching.

VII. Conclusions and Future Work

In this paper we presented the latest release of OpenDwarfs, which provides enhancements upon the original OpenDwarfs benchmark suite. We verified functional portability of dwarfs across a multitude of parallel architectures and characterized a subset's performance with respect to specific architectural features. Computation and communication patterns of these dwarfs lead to diversified execution behaviors, thus corroborating the suitability of the dwarf concept as a means to characterize computer architectures. Based on dwarfs' underlying patterns and profiling we were able to provide insights tying specific architectural features of different parallel architectures to such patterns exposed by the dwarfs.

Future work with respect to the OpenDwarfs is multi-faceted. We plan to:



(a) (b)

Fig. 6: (a) NW profiling on HD7660D. (b) BFS cache performance comparison between HD7970 and HD7660D.

(a) Further enhance the OpenDwarfs benchmark suite by providing features such as input dataset generation, automated result verification and OpenACC implementations. More importantly, we plan to *genericize* each of the dwarfs, i.e., attempt to abstract them on an even higher level, since currently some dwarf applications may be considered too application-specific.

(b) Characterize more modern parallel architectures, including Altera FPGAs by using the Altera OpenCL SDK and evaluate different vendors' OpenCL runtimes. Further characterization with input datasets of varying size and even shape is a potential future research avenue.

(c) Provide architecture-aware optimizations for dwarfs, based on the existing naïve implementations. Such optimizations could be eventually integrated as compiler back-end optimizations after some form of application signature (i.e., dwarf) is extracted by code inspection, user-supplied hints, or profile-run data.

ACKNOWLEDGMENTS

This work was supported in part by the Institute for Critical Technology and Applied Science (ICTAS) at Virginia Tech and by EU (European Social Fund ESF) and Greek funds through the operational program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. The authors would also like to thank the OpenDwarfs project, supported by the NSF Center for High-Performance Reconfigurable Computing (CHREC) via NSF I/UCRC Grant IIP-1266245.

REFERENCES

[1] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, Salt Lake City, UT.

[2] *Implementing FPGA Design with the OpenCL Standard*, 2nd ed., Altera Corporation, 2012.

[3] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, New York, NY.

[5] W.-C. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 Dwarfs: A Work In Progress," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, Boston, MA.

[6] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[7] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using Benchmarking to Advance Research: a Challenge to Software Engineering," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, Washington, DC.

[8] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects," in *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLoP '10)*, New York, NY.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, Austin, TX.

[10] 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*, Pittsburgh, PA.

[11] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, New York, NY.

[13] *Altera SDK for OpenCL: Programming Guide*, Altera, 2013. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA.