MachSuite: Benchmarks for Accelerator Design and Customized Architectures

Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, David Brooks
Harvard University, Cambridge, MA, USA
{reagen, rdadolf, shao, guyeon, dbrooks}@eecs.harvard.edu

***Abstract*—Recent high-level synthesis and accelerator-related architecture papers show a great disparity in workload selection. To improve standardization within the accelerator research community, we present *MachSuite*, a collection of 19 benchmarks for evaluating high-level synthesis tools and accelerator-centric architectures. MachSuite spans a broad application space, captures a variety of different program behaviors, and provides implementations tailored towards the needs of accelerator designers and researchers, including support for high-level synthesis. We illustrate these aspects by characterizing each benchmark along five different dimensions, highlighting trends and salient features.**

I. Introduction

With the demise of Dennard scaling and inexorable advance of Moore's Law, today's architects are confronting chips filled with more transistors than can be fully powered. This phenomenon has spurred a flurry of research into new mechanisms that promise to continue scaling performance while on a budget. One direction showing promising efficiency gains are hardware accelerators: fixed-function hardware blocks which compute a specific task at a fraction of the cost of a general-purpose processor. As a result, there is rising interest in methods for the design and integration of accelerator components.

Designing hardware accelerators in RTL languages like Verilog or VHDL is widely acknowledged to be time-consuming and complicated. To improve productivity, the CAD community introduced High-Level Synthesis (HLS) tools, which automatically synthesize RTL code from a high-level language like C or C++. Modern HLS frameworks can produce code with performance on par with hand-written RTL for some workloads. However, as the field matures, it has faced new challenges from more complicated, irregular applications.

The interest in hardware specialization has brought about challenges at the architecture level as well. Fine-grained heterogeneity has appeared in processors, and the rising popularity of systems-on-chip (SoCs) in the mobile domain is making hardware accelerators commonplace. Architects now have to contend with a much larger range of on-chip interactions, and with so many different possible tactics and mechanisms, it is a daunting task to choose the right design direction.

These trends are two sides of the same coin: advances in accelerator-centric research have caught us unprepared to quantitatively and objectively evaluate the relative strengths and weaknesses of such a diverse collection of techniques. Much of this can be ascribed to a simple lack of standardization. A survey of recent publications (Section II-A) involving hardware accelerators revealed that of the 88 distinct benchmarks used across 25 papers, 64 of them were only ever used *once*, as shown in Figure 1. The most popular benchmarks appeared in less than half of those. But the problem is deeper than that. The wide variety of architectural approaches taken by different researchers exacerbates this sensitivity to algorithmic changes. Even the same kernel written using two different algorithms, I.e., merge verses radix sort, can produce substantially different behaviors (Section II-B). Furthermore, benchmarks for accelerator designs, especially those used by HLS tools, have different needs than those for traditional general purpose processors. Implementation style, tool directives and pragmas, and even basic code cleanliness can cause non-trivial performance artifacts (Section II-C). We need a better benchmark suite for standardization, commensurability, and quantitative evaluation.

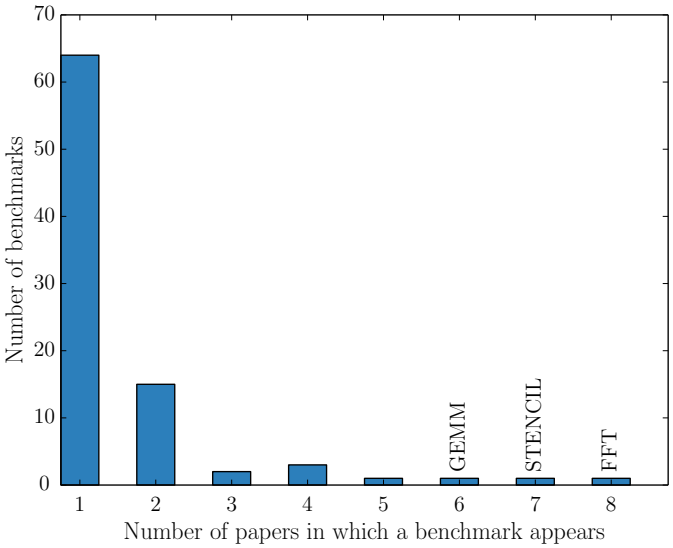


Fig. 1: The frequency distribution of benchmarks used in 25 papers. The vast majority appear only once.

Consequently, we present MachSuite: a new accelerator-centric benchmark suite tailored to the needs of both the HLS and architecture communities. MachSuite is a set of 19 benchmarks spanning 12 different kernels, written to cover a diverse set of application domains and to incorporate distinct algorithmic choices. All the benchmarks in MachSuite are HLS synthesizable, providing architecture researchers an easy way to quickly generate a diverse set of hardware accelerators. In this paper, we explain the rationale for our design choices (Section III), provide a description for every benchmark (Section IV), demonstrate diverse workload behaviors across benchmarks (Section V), and discuss its applicability in accelerator-related research (Section VI.)

II. The Need for a New Benchmark Suite

The motivation for MachSuite was derived from the observed lack of standard, well-defined benchmarks for hardware accelerators. In this section, we draw a distinction between benchmark kernels, algorithms, and implementations and discuss the importance of standardization across each of them. We then examine existing, related benchmarks and show why they are unsuitable for accelerator-centric research.

A. The Need for Standard Kernels

When it comes to choosing benchmarks for quantitative evaluation, researchers look for clear, effective demonstrations of their contributions. Absent any established standard, they choose the most appropriate solution given their situation. Unfortunately, the wide variety of different approaches often leads to divergence in the choice of benchmarks used. Without the ability to make direct, objective comparisons, it becomes increasingly difficult to understand where or when fundamental advances are being made.

To understand the state of the accelerator community, we surveyed 25 publications from top architecture and CAD conferences over the last several years. We found that of the 88 different benchmarks used, only 24 appear in more than one paper. Figure 1 shows the distribution of the benchmarks we collected, with citations to the publications in Table I. While commonalities exist, even the three most frequent benchmarks (GEMM, stencil, and FFT) showed up in less than a third of the papers surveyed. A result suggesting that even though the community continues to improve accelerator design tools and architectures, only a small subset of results can be quantitatively compared.

B. The Need For Standard Algorithms

A more subtle issue is differences between which algorithms are used to solve a particular kernel. A program that uses a blocked, in-place method instead of a recursive, out-of-place one will behave very differently, even when solving the same problem. We do not want to discourage the use of different approaches. It is important for the community to be able to study and compare different algorithms, but we need transparency.

A recent case study evaluating k-means clustering on an FPGA using HLS demonstrates this discrepancy [51]. The authors wrote C code for two different algorithms and synthesized RTL for each. While these programs solved the same problem with the same inputs, the performance gap was $1.6\times$–$12.8\times$ between them. While this should not come as a surprise to experienced developers, it reinforces the fact that without agreement on a common algorithm, it is easy to mask fundamental distinctions between research results.

C. The Need For Standard Implementations

In a similar vein, differences in coding style and implementation can cause significantly different results. For instance, writing a vector of structures instead of a structure of vectors has consequences on locality. While this effect shows up in benchmarks for conventional platforms, accelerators are more sensitive to these changes, especially designs produced using HLS tools.

Research Area	Publications
Acc-Centric Architectures	[11], [12], [13], [14], [15], [16], [21] [22], [34], [37], [40], [41], [43], [46]
HLS Optimizations	[4], [6], [7], [17], [20], [23], [31], [32], [33], [35], [48]

TABLE I: List of papers used to construct Figure 1.

One recent example involved a parametric sweep over a space of implementations of the same kernel and algorithm [43]. The authors showed that by changing only the organization of the inner loop of a parallel scan code, the Pareto-optimal points for power and performance can vary by an order of magnitude, even when given similar hardware resources, memory bandwidth, and parallelism.

D. Existing HLS Benchmark Suites are Unsuitable

To the best of our knowledge, CHStone [24] is the only existing HLS benchmark suite. Designed to be an evaluation mechanism for HLS tools, CHStone focuses on a small number of low-level computations. While useful for evaluating the capabilities of older HLS frameworks, it falls short when put in the context of the complex designs handled by modern CAD toolchains and the expansive system designs that many architects are contemplating and constructing.

CHStone is comprised of 12 benchmarks: 4 arithmetic operators, 3 cryptographic functions, 4 multimedia components, and a simplified MIPS processor. While these benchmarks go a small way to providing proxies for some accelerator domains, the suite as a whole misses many quintessential themes. For instance, the three most prominent benchmarks found in the literature survey (GEMM, stencil, and FFT) do not appear in CHStone. Moreover, CHStone really only captures two classes of kernels: cryptography and media processing. We believe more diverse workloads are needed.

E. Existing GPU Benchmark Suites are Unsuitable

Many high-quality benchmarks have emerged from the GPGPU community in the last several years [9], [18], [44]. These suites offer a wide variety of benchmarks designed to exercise the myriad architectural features found on modern graphics cards. However, GPUs have a very specific design envelope.

Large numbers of vector processing units argue for a data parallel programming paradigm; a deep, complicated memory hierarchy demands attention at the algorithm level; the split main memory system strains application writers. Most GPU benchmarks do an admirable job of contending with these constraints, but accelerator-centric architectures have a completely different design space, one characterized by many more degrees of freedom. Control flow can either be a non-issue or a challenge; memory can either be incredibly fast and cheap (via custom scratchpad structures) or complicated and expensive (if many accelerators are sharing a coherent cache); system integration can be anywhere from inside a larger processor pipeline to attached to an on-chip network to a fully independent, heterogeneous collection. While GPU benchmarks satisfy a particular application space, HLS users

and architects studying accelerators need a separate set of benchmarks tailored for their needs.

III. The Design of the MachSuite Benchmarks

To address these gaps, we created MachSuite, a set of 19 benchmarks covering 12 different application kernels. In this section we present an overview of MachSuite's features and how we arrived at the design decisions we made.

A. Kernel Selection

The MachSuite workloads were chosen to satisfy two basic criteria: diversity and coverage. Diversity is a measure of the similarity of kernels to each other; we would like to ensure that each kernel brings something new to the table. Coverage is a measure of representativeness with respect to the field; we would like at least one kernel to be similar to any application a given user has.

To achieve diversity, we used a battery of workload characterization metrics to judge whether any two programs had overly similar execution behavior. We present a quantitative characterization of each benchmark in Section V to evaluate workload diversity.

To achieve coverage, we first looked at our literature survey. Directly including every code in that list would be prohibitive, but we *can* match each benchmark published more than once with a similar one of our benchmarks, ensuring that the behavior of MachSuite encompasses workloads the community cares most about. This only determines coverage of research that has already been done. We augmented our set of kernels to include additional programming patterns that provide new targets for accelerator designers and system architects to evaluate against. In Table II, each benchmark is described and assigned to an application pattern to show, at a high level, the application space covered by MachSuite.

B. Algorithm Selection

For each kernel, we select an algorithm representative of the tactics commonly used to solve it. While we could have chosen cutting-edge methods and invested heavily in heuristics for our benchmarks, we elected to favor simplicity over optimality for two reasons. First, optimality is not portable, and an algorithm which performs well given one set of assumptions may flop when run in another context. Codes with this "sometimes-optimal" flavor are rarely useful as benchmarks. Secondly, simplicity is a virtue for experimentation. Benchmark suites are continually used and abused in ways their designers never predicted. MachSuite embraces this spirit of creativity, we expect our codes will be examined and enhanced to address new challenges.

For some kernels, there are several common algorithms in use, and we feel this is an opportunity. For 7 of our kernels, we provide two distinct algorithms which solve effectively the same problem in different ways or with different characteristics. This not only serves to provide better diversity in program behavior but it also has the interesting side effect of allowing direct comparison at the algorithm level. This provides insight into which types of methods a particular research tool or technique is more amenable to.

Kernel/Algorithm	Description	Berkeley Dwarf [5]
AES/AES	AES encryption	Combinational logic
BACKPROP/BACKPROP	Neural network training	Unstructured grids
BFS/BULK	Breadth-first search	Graph traversal
BFS/QUEUE	Breadth-first search	Graph traversal
FFT/STRIDED	Fast Fourier transform	Spectral methods
FFT/TRANSPOSE	Fast Fourier transform	Spectral methods
GEMM/NCUBED	Matrix multiplication	Dense linear algebra
GEMM/BLOCKED	Matrix multiplication	Dense linear algebra
KMP/KMP	String matching	Finite state machines
MD/KNN	Molecular dynamics	N-body methods
MD/GRID	Molecular dynamics	N-body methods
NW/NW	DNA alignment	Dynamic programming
SORT/MERGE	Sorting	Map reduce
SORT/RADIX	Sorting	Map reduce
SPMV/CRS	Sparse matrix/vector multiplication	Sparse linear algebra
SPMV/ELLPACK	Sparse matrix/vector multiplication	Sparse linear algebra
STENCIL/STENCIL2D	Stencil computation	Structured grids
STENCIL/STENCIL3D	Stencil computation	Structured grids
VITERBI/VITERBI	Hidden Markov model estimation	Graphical models

TABLE II: The MachSuite benchmarks.

C. Inputs Values and Size

Benchmark inputs are an oft-overlooked feature. Like most packages, MachSuite includes standard inputs and reference outputs for each benchmark, including simple code to automatically verify correctness. Unlike many other packages, we also recognize the importance of realistic input data for certain types of algorithms. For regular, compute-driven programs, randomized inputs suffice as the execution behavior is invariant with respect to the input values. For data-driven algorithms like BFS/QUEUE and SPMV/CRS, the shape and structure of the input can actually dictate runtime characteristics like locality and branch behavior. MachSuite provides input generators for these programs and tunes parameters to reflect more realistic inputs.

We also recognize that input size matters, not only for ease of use but for accurate results. If a researcher is running two benchmarks with memory footprints of 1KB and 1GB, there is little hope that a meaningful conclusion can be drawn by comparing the two. We strive for a reasonable degree of homogeneity amongst all 19 of our benchmarks. To achieve this, we attempt to control the input and algorithmic parameters of each code such that the maximum memory footprint is around 32KB—an estimate of the average size of an L1 cache.

While we realize this number is somewhat arbitrary (choosing half or twice that amount would not change the flavor of our benchmark suite), it is important to pick *some* number as a target. For benchmarks with little to no working sets (I.e. AES/AES), we don't artificially try to inflate this number. It is reasonable to ask whether constraining our benchmarks to KB instead of MB or GB is limiting. MachSuite's behavior will not accurately reflect memory characteristics at these scales, but we also believe that if researchers are interested in building scalable accelerator-centric systems, MachSuite's kernels supply a practical evaluation of the innermost building blocks. While the design and characterization challenges associated with building a composable, scalable system are intriguing, they also fall outside of the scope of the MachSuite benchmarks, and we leave that problem open for future work.

D. Code Considerations

We made two complementary design choices with respect to programming style: first, we use clean, readable, and modifiable code, and second, MachSuite is amenable to HLS. Clean code is an enabling factor. Many accelerator researchers study, dissect, and extend programs as a matter of course, whether it is through restructuring, annotation, or static analysis. As a result, code written in an accessible style becomes a non-negligible advantage in terms of time-to-solution.

Additionally, since HLS users constitute an important component of MachSuite's target audience, it is critical that our benchmarks work seamlessly with HLS workflows. We achieve this by using a constrained subset of C and by supplying HLS directives to enable RTL synthesis. None of the data structures in MachSuite are heap allocated, we remove non-array pointer arithmetic, we avoid recursive function calls, and when possible, we prefer finite upper bounds on parameters. These concessions are reflections of the practical realities of hardware programming; while some HLS tools may be able to handle these cases, the resulting synthesized RTL is often a worst-case estimate and unrealistic.

Directives are a cornerstone of many HLS tools [36]. In MachSuite, we label each loop and provide scripts so users can generate a particular configuration or explore the space of possible solutions. While we make a reasonable attempt to craft directives which will produce efficient hardware, we don't claim they are optimal. Performance is dependent on many external factors, including system-level assumptions, choice of HLS tool, and underlying hardware components, so the dream of a single, perfect set of directives is unrealistic.

E. Limitations

MachSuite is not without limitations. As noted in III-C, the size of the inputs to the benchmarks is small. Users looking to stress memory hierarchies and design more complex systems may find the current set of inputs too small to provoke interesting interactions. This version of MachSuite was developed for datapath design and only looks at the first level of the memory hierarchy; we are currently working on scaling MachSuite's input sets to larger scales such that a wider range of researchers will find them useful. Additionally, while we have made an attempt to capture diversity in both application domains and code characteristics, we do not provide an absolute metric of benchmark similarly. Our characterization (Section V) is a start, but we have not yet carried out the deeper scrutiny afforded to mature benchmark suites like SPEC [38] or Rodinia [10].

IV. Benchmark Descriptions

AES/AES

The Advanced Encryption Standard [39] is an iterated block cipher designed in 1998 as a replacement for the DES algorithm. The core computation is a series of alternating substitution and permutation phases on a 16-byte state matrix. AES tends to be amenable to both hardware and software implementations, largely due to its parallelizability, use of byte-oriented arithmetic operators, and transformations small enough for lookup tables. MachSuite's implementation of AES is adapted from Ilya Levin [29] and provides a lookup table optimization for the primary S-box.

BACKPROP/BACKPROP

Neural networks are a widely used machine learning technique, with applications including classification, pattern recognition, and control theory. Based on a mathematical model similar to their biological counterparts, neural nets are multi-layered, feed-forward networks with variable weights on every dependence edge. Using a neural network to perform a specific task is usually computationally simple: input is simply fed through the network and outputs observed. Training a neural net is more expensive, involving iteratively tuning a large number of parameters to fit a given training set. Backpropagation is a common training algorithm which takes differences between the output of an untrained net and the desired output and pushes them backwards through the network, updating node weights proportionally as it goes.

BFS/BULK

Breadth-first search is a fundamental building block for many graph algorithms, including path finding, network flow, and community detection. Usually expressed as a sequence of expanding "horizons" or "frontiers" of nodes, BFS is notable for being memory intensive and having irregular-but-plentiful parallelism. The BFS/BULK code uses a brute-force, data parallel method [26], [25] which is typically used on SIMD and vector architectures. Additionally, the execution behavior of BFS is heavily dependent on the structure of its input graph, and mesh or Erdös-Rényi graphs often underestimate memory hot-spotting and overestimate typical graph diameter. MachSuite provides a low-diameter, scale-free graph using the R-MAT algorithm [8].

BFS/QUEUE

BFS can also be computed using a work queue algorithm which dynamically tracks the current horizon. Queue-based codes typically trade off lower memory bandwidth requirements for increased bookkeeping. The BFS/QUEUE variant produces an identical solution to BFS/BULK but exhibits a different node traversal order, and, by extension, notably different computational characteristics.

FFT/STRIDED

The Fast Fourier Transform is a ubiquitous kernel with applications in almost every field. It is the most commonly used kernel in the literature we surveyed. The canonical Cooley-Tukey "butterfly" method we use is characterized by a wide range of strided access patterns and nested, triangular loop structures. We provide a straightforward, iterative implementation of a 1024-point, complex FFT.

FFT/TRANSPOSE

Due to the nonuniform memory costs of most modern architectures, a common optimization for the FFT is to compute a series of small-radix FFTs interleaved with transpose operations. This technique trades off data manipulation overhead for improved locality and the opportunity to optimize for a single, fixed-size FFT. MachSuite provides a 512-point, complex FFT that uses an 8-point small-radix FFT, adapted from the inner loop of a well-tuned GPU code [47].

GEMM/NCUBED

Matrix multiplication is likely the most re-used building block

in numerical software and a cornerstone of any linear algebra package. Sporting high computational density, extremely regular behavior, and an easily manipulated mathematical structure, it is a common target for automatic- and hand-tuning. GEMM/NCUBED is a naive, $O(N^3)$ implementation provided as a well-understood reference point.

GEMM/BLOCKED

Matrix multiply is more commonly computed using a blocked loop structure. Commuting the arithmetic to reuse all of the elements in one block before moving onto the next dramatically improves memory locality. MachSuite's version uses a fixed blocking factor of 8 and is based on the algorithm proposed in [28].

KMP/KMP

String matching finds applications in everything from packet filters to scientific codes to desktop applications. The Knuth-Morris-Pratt algorithm is a fast string matching technique which appeared in the late 1970s [27]. The key improvement in KMP was a small, precomputed data structure enabling the algorithm to skip ahead in the input string when a mismatch is discovered. KMP/KMP implements both the matching and precomputation steps, and we supply an English-text input array which matches a short pattern with low frequency.

MD/KNN

Molecular dynamics simulations are a class of n-body problems which underpin most computational chemistry packages. While most MD codes include a variety of iterated equations, the dominant component is normally the calculation of long-distance forces, which is order $O(n^2)$. Our two MD benchmarks both compute Lennard-Jones potentials, a common approximation to the van der Waals interactions between all pairs of atoms. Since the strength of these interactions die out as a sixth-order polynomial function of distance, most simulations further approximate the force calculation by only considering nearby pairs of points. MD/KNN (short for k-nearest neighbors) uses explicit, fixed-length neighbor lists to track the relevant molecular interactions. The code itself is adapted from the SHOC implementation [18].

MD/GRID

Another variant of MD, used by many scalable computational chemistry packages [2], [3], replaces explicit neighbor lists with a three-dimensional grid. Force calculations are instead computed on all particles in the current and adjacent grid cells. This technique pays a price in bookkeeping overhead to track and iterate over grid cells, but it improves memory locality and enables memory partitioning. MachSuite's MD/GRID and MD/KNN codes use the same input set and agree within 0.1%.

NW/NW

The Needleman-Wunsch algorithm is DNA alignment technique first introduced in 1970. The method is a canonical dynamic programming problem that optimizes a similarity score between two strings. Our implementation is a wavefront computation that populates a square similarity matrix as it executes. Materializing the score matrix, while expensive, allows reconstruction of the optimal alignment.

SORT/MERGE

Sorting, while a useful kernel in its own right, also serves as a building block for many other algorithms. Merge sort is an out-of-place algorithm which tends to be popular on parallel platforms due to its simple structure and fewer data dependencies. MachSuite includes an iterative implementation of a 4096 integer sort.

SORT/RADIX

Radix sort is a typical non-comparison-based sort, often used when handling input sets with small value ranges and in parallel contexts. Non-comparison sorts exploit properties of the value domain (bit-pattern regularity, in the case of radix sort) to lower computational complexity. Most radix sort codes utilize a highly tuned, fixed-size inner loop which aligns with convenient hardware performance breakpoints. MachSuite's version is adapted from a radix-4 integer sort [18].

SPMV/CRS

Sparse methods leverage the annihilating property of zeroes in linear systems to omit irrelevant computation. Sparse matrices often appear when solving systems of dependent equations or computing properties on high-diameter graphs. The core calculation of sparse matrix/vector multiplication is identical to the dense version, but organizing and tracking the nonzero elements dramatically changes the computational characteristics. SPMV/CRS uses compressed row storage format for the nonzero elements [30]. Since sparse matrix operations often depend heavily on the structure and density of the input matrix, we provide an electrical admittance matrix from the classic Harwell-Boeing matrix collection [1] as a proxy for the behavior of an iterative solver.

SPMV/ELLPACK

Ellpack is an alternative to contiguous nonzero packing, originally introduced as an external storage format [49] but recently rediscovered for vector architectures [45]. Ellpack trades off increased storage overhead for regularity in access pattern, padding out nonzero arrays to a fixed length to enable sequential access.

STENCIL/STENCIL2D

Stencil codes are a core component of many computer vision applications and scientific simulations. The inner loop of a stencil code is a fixed-size computational template which is moved across a large input grid. The size and shape of the stencil itself varies across applications. The majority of codes use a stride-1 motion across the data. MachSuite contains a basic, 9-point, 2D stencil [18].

STENCIL/STENCIL3D

Stencils generalize to different grid configurations and dimensions. However, as the grid changes, so do the execution characteristics. To demonstrate the difference between surface and volume stencils, we also provide a 7-point, 3D stencil. The STENCIL/3D benchmark is based on the algorithm proposed in [19].

VITERBI/VITERBI

Hidden Markov models are a widely used stochastic model with applications ranging from information coding to speech recognition to bioinformatics. The Viterbi algorithm is a dynamic programming method which computes the most likely Markov chain based on a set of observations and a pair of probability matrices. In contrast to Needleman-Wunsch, another dynamic programming code, Viterbi exhibits higher

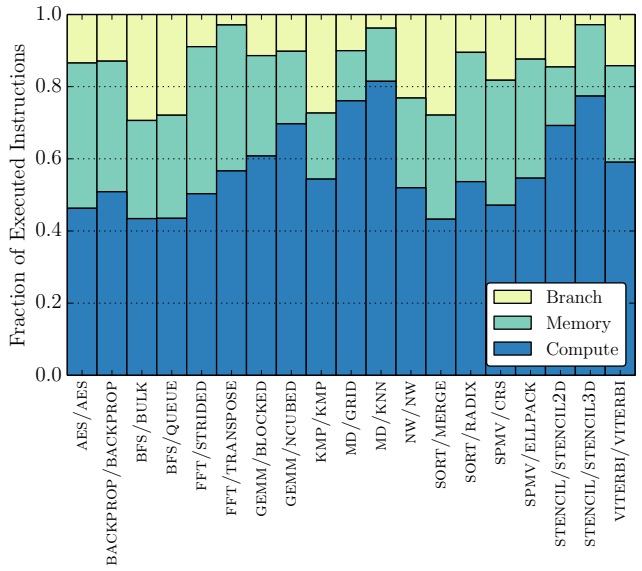


Fig. 2: Instruction Mix

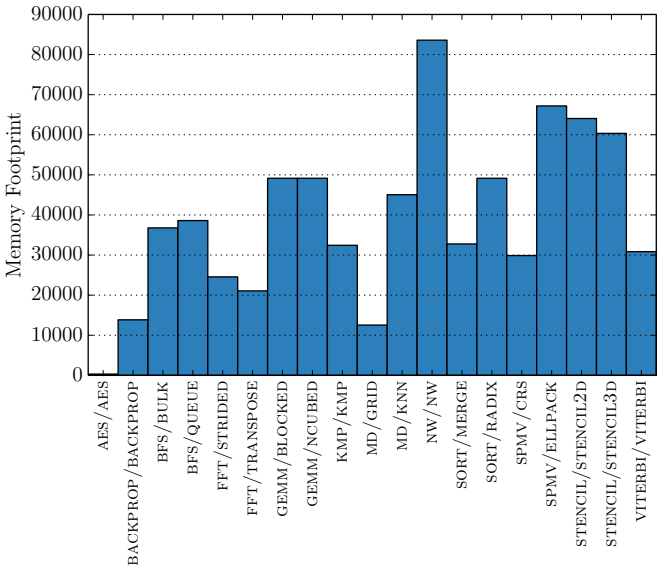


Fig. 3: Unique Memory Bytes

computational density– dense transition matrices induce combinatorial update steps.

V. Characterization

To demonstrate the wide variety of behaviors in MachSuite, we subject it to a battery of quantitative characterizations. These tests, ranging from operation counts to memory pattern analysis to entropy calculations, are intended to provide insight into the overall properties of the MachSuite benchmarks, but also into the way those properties interact with specialized hardware.

MachSuite is targeted for fixed-function accelerator design– a compute paradigm in which no ISA exists. To circumvent this, we use Workload ISA-Independent Characterization (WIICA) [42], a framework targeted for custom architecture, to characterize the behavior of MachSuite. WIICA instruments the program to emit a dynamic instruction trace with program runtime information, such as memory reference addresses and branch decisions. After the dynamic trace profiling, WIICA applies a set of workload analysis. To demonstrate MachSuite's amenability to HLS, we synthesize each benchmark by running it through Vivado HLS.

A. Instruction Mix

Breaking down a program into compute, memory, and control flow instructions (Figure 2) is a good first-order measure of a program's leanings. While memory instructions are relatively consistent, 20–30% of MachSuite's operations, control flow displays much more variance. The extreme examples (KMP/KMP, BFS/BULK, and BFS/QUEUE) are all heavily conditional-laden.

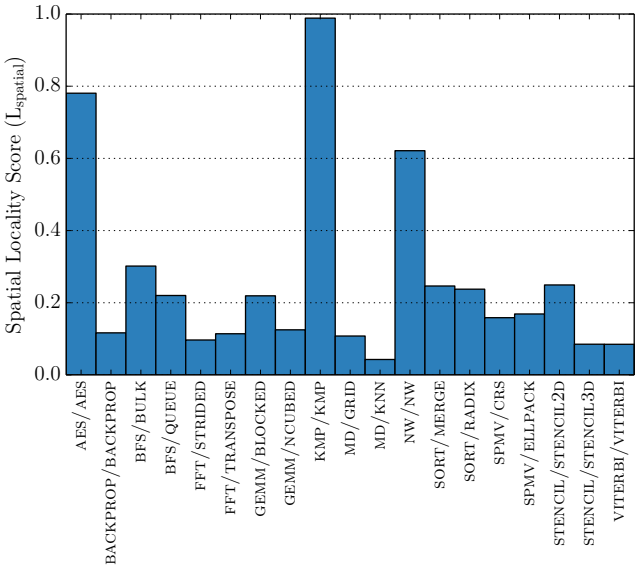
The differences in SPMV/CRS and SPMV/ELLPACK memory and branch instruction percentages reflect the differences in the two algorithms. SPMV/CRS requires less memory operations but more branching while SPMV/ELLPACK pads the compressed matrix with zeros, lowering control overhead at the expense of more memory traffic.

With respect to hardware accelerators, ideal workloads are typically comprised mostly of compute instructions. High percentages of branch instructions may or may not be beneficial, depending on the branch patterns. High predictability means control is easily transformed to fixed data paths in hardware, eliminating overhead and bloat. Low predictability results in multiple execution paths and explicit logic to handle control, demanding more design effort to understand and implementation trade-offs such as predicated execution. (Branch predictability is investigated further in Section V-D.)

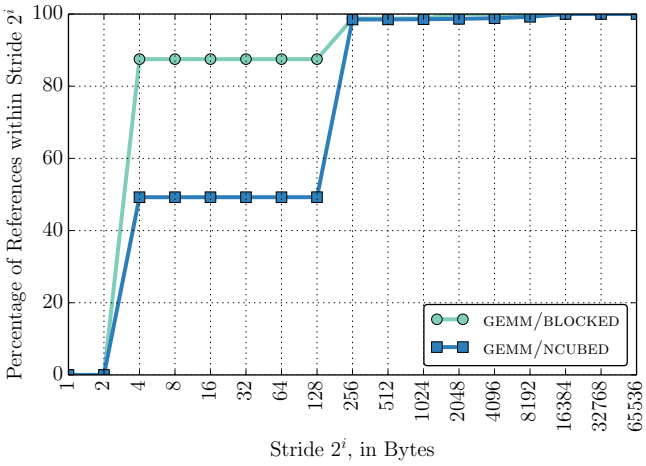
B. Memory Footprint

Memory footprint (Figure 3) is a measure of the total number of unique bytes a benchmark addresses. Most programs fit within 16 KB and 64 KB, close to the established 32 KB target in Section III-C. AES/AES is an obvious exceptions, as it requires almost no state storage nor input data. Another exception, MD/GRID, is actually caused by its algorithmic counterpart, MD/KNN. In order to enable direct comparisons, we use the same input and force calculations on both molecular dynamics codes, and since MD/KNN has a large auxiliary storage array, we reduce the input grid on both versions until the latter fits in a reasonable size. The end result is that MD/GRID has a much smaller footprint. At the other end of the spectrum, NW/NW utilizes over 80 KB of space as a result of the $O(n^2)$ array which stores its wavefront computation. Shrinking the DNA input strings could have alleviated this, but it might also have biased the input characteristics unrealistically.

One interesting point of comparison in the memory footprint plot is the difference in behavior between SPMV/CRS and SPMV/ELLPACK. SPMV/ELLPACK has over twice the number of memory accesses as its counterpart, despite computing the same solution. Inspecting the code reveals that this is a result of the algorithm, not an artifact. In order to vectorize its inputs, SPMV/ELLPACK pads out each row of its nonzero matrix to the maximum length of any row. While this provides very good regularity and stride access pattern, it also pays a price in having to store additional zeros.



(a) Spatial Locality Score



(b) SPMV

Fig. 4: Spatial Locality.

C. Spatial Locality

Spatial locality (Figure 4a) measures where a program's memory accesses occur in relation to each other. This property is important for accelerator design as it can enable optimizations like prefetching and burst transfers. To summarize the spatial locality across MachSuite, we use a simple metric defined by Weinberg et al. [50]:

$$\mathrm{L}_{\mathrm{spatial}} = \sum_{\mathrm{stride}=1}^{\mathrm{stride}<\infty} \frac{P(\mathrm{stride})}{\mathrm{stride}}$$

The intuition behind this plot can be explained by two phenomenon. First, programs with large amounts of stride-one code have very high spatial locality. This is an expected result. The second, slightly more subtle, contributing factor is that the primary datatype radically affects this metric. Byte-oriented programs (KMP/KMP and NW/NW) have tightly packed arrays which they iterate over. Programs using double-precision floating point numbers (e.g., FFT/STRIDED and MD/KNN) can only achieve a minimum stride distance of 8 bytes.

To better understand the relationship between algorithms and spatial locality, we look at the cumulative fraction of memory references which occur at a given stride for GEMM/BLOCKED and GEMM/NCUBED, shown in Figure 4b. The initial jump in both benchmarks at 4 bytes is a result of stride-one accesses of 32-bit integers from a matrix row in the innermost loop. In these inner loops, both benchmarks have two memory references: GEMM/NCUBED reads two elements from the input matrices, and GEMM/BLOCKED reads one element from an input matrix (the other resides in a temporary variable) and writes one element to a temporary output block. The larger percentage of local references in GEMM/BLOCKED shown in the plot is due to the fact that the input matrix and output matrix are both being accessed with stride-one. In GEMM/NCUBED, only the row elements are read consecutively—the column elements are 256 bytes apart. As a result, only half as many of GEMM/NCUBED's memory references are captured by small strides. The second jump at a stride of 256 accounts for this in GEMM/NCUBED. Likewise, in GEMM/BLOCKED, each block is 256 bytes of data, so the second jump is a result of a middle loop moving on to the next block.

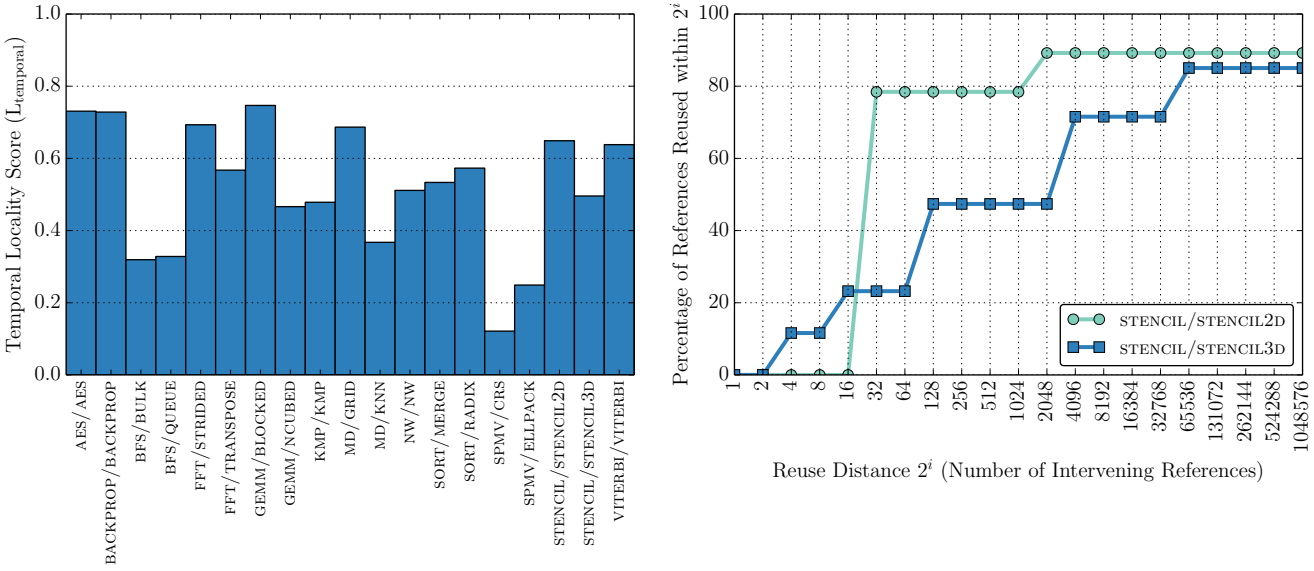
D. Temporal Locality

Temporal locality (Figure 5a) measures the number of memory accesses that occur before any given address is repeated. It is a measure of re-use, and it is typically exploited in hardware by adding registers or caches to store data nearby so it can be cheaply re-fetched. Weinberg et al.[50] introduce a summary metric for temporal locality:

$$\mathrm{L}_{\mathrm{temporal}} = \sum_{i=0}^{i<\log_2(N)} \frac{((\mathrm{dist}_{2^{(i+1)}} - \mathrm{dist}_{2^i}) * (\log_2(N) - i))}{\log_2(N)}$$

Benchmarks with lower temporal locality typically either stream elements of a data structure, accessing them only once (i.e. KMP/KMP) or exhibit irregular accesses that have low probability of reuse (i.e. BFS/BULK).

The STENCIL/STENCIL2D and STENCIL/STENCIL3D algorithms present an interesting contrast. STENCIL/STENCIL2D applies a 3×3 filter to a large input array. Of the 9 data elements loaded for each computation, 6 are reused in the next loop iteration, and in combination with the 9 reused elements of the filter, are reused approximately 80% of the memory references every time we shift the stencil over. The jump after 1024 comes from wrapping around—each row is 256 integers, or 1024B. If we contrast this pattern with the 3D stencil, we see a similar structure. STENCIL/STENCIL3D uses a 7-point, star-shaped filter, but the filter parameters are inlined, not held in a temporary array. This difference, plus a rearranged filter order, accounts for the pair of plateaus at reuse intervals of 4 and 16. We can also clearly see both the row- and column-wraparound at 128 and 4096, respectively. Due to the shape of the stencil, the bump at 65536 is actually the "reuse" of the



(a) Temporal Locality

(b) Stencil Reuse Distance

Fig. 5: Temporal Locality.

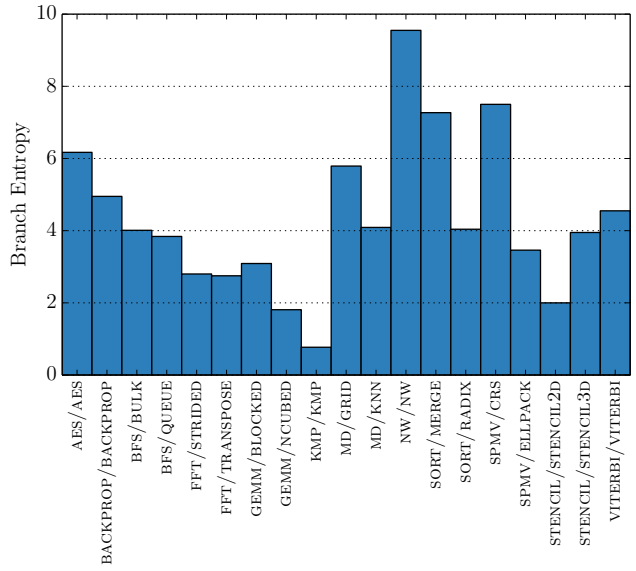


Fig. 6: Branch Entropy

last filter element with the initialization of the array. Neither stencil code is computed in-place, so the output arrays (which have no temporal reuse) cause both to saturate around 85%.

E. Branch Entropy

Branch entropy [52] is a measure of control flow regularity. It works by computing the information content contained in all 16-long sequences of branching decisions made by the program. Figure 6 shows the branch entropy of each MachSuite workload. Algorithms with long, fixed trip counts (SPMV/ELLPACK, STENCIL/STENCIL2D) have almost no entropy, while irregular control flow (SPMV/CRS) and data-dependent branching (NW/NW, SORT/MERGE) result in a higher branch entropy.

To understand how different benchmark's entropy scores reflect their behavior, we consider SORT/MERGE and SORT/RADIX. SORT/MERGE has over 3 bits more entropy than SORT/RADIX, implying that its control regularity has much more randomness. SORT/RADIX has no dependencies on input data. It bins inputs, then rearranges the input based on the cardinality of each resulting bucket. In contrast, SORT/MERGE has a control flow almost exclusively driven by input values. While the divide-and-conquer approach is regular, the reconstruction of the array branches at every step conditioned on the input's value.

Where SORT/RADIX trumps SORT/MERGE in regularity of control structure SORT/MERGE beats SORT/RADIX in performance (Figure 7). Trade-offs like this enable designers and researchers to explore and optimize algorithms to best match their design target.

F. Synthesis Results

To demonstrate that MachSuite's benchmarks are synthesizable, each is run through Vivado HLS. FPGA resource requirements including flip-flops (FF), look-up-tables (LUT), and digital-signal-processor slices (DSP) are shown in Figure 7. We also report performance results including cycle counts and execution time. For synthesis, no optimizations or directives were applied.

FFT/TRANSPOSE is a prime candidate for optimization. Within each benchmark exists a vast design space MachSuite users can explore and optimize. Without guidance, Vivado synthesizes distinct logic each time a function is invoked. However, FFT/TRANSPOSE has distinct program phases, alternating between the execution of smaller FFTs and reorganizing inputs and partial products by shuffling arrays that are then the inputs to the same small FFT computations. Having non-overlapping phases ensures the FFT and shuffling logic can be shared, reducing the number of required instantiations of these functions without affecting performance.

In Figure 7 the variance of execution times of different

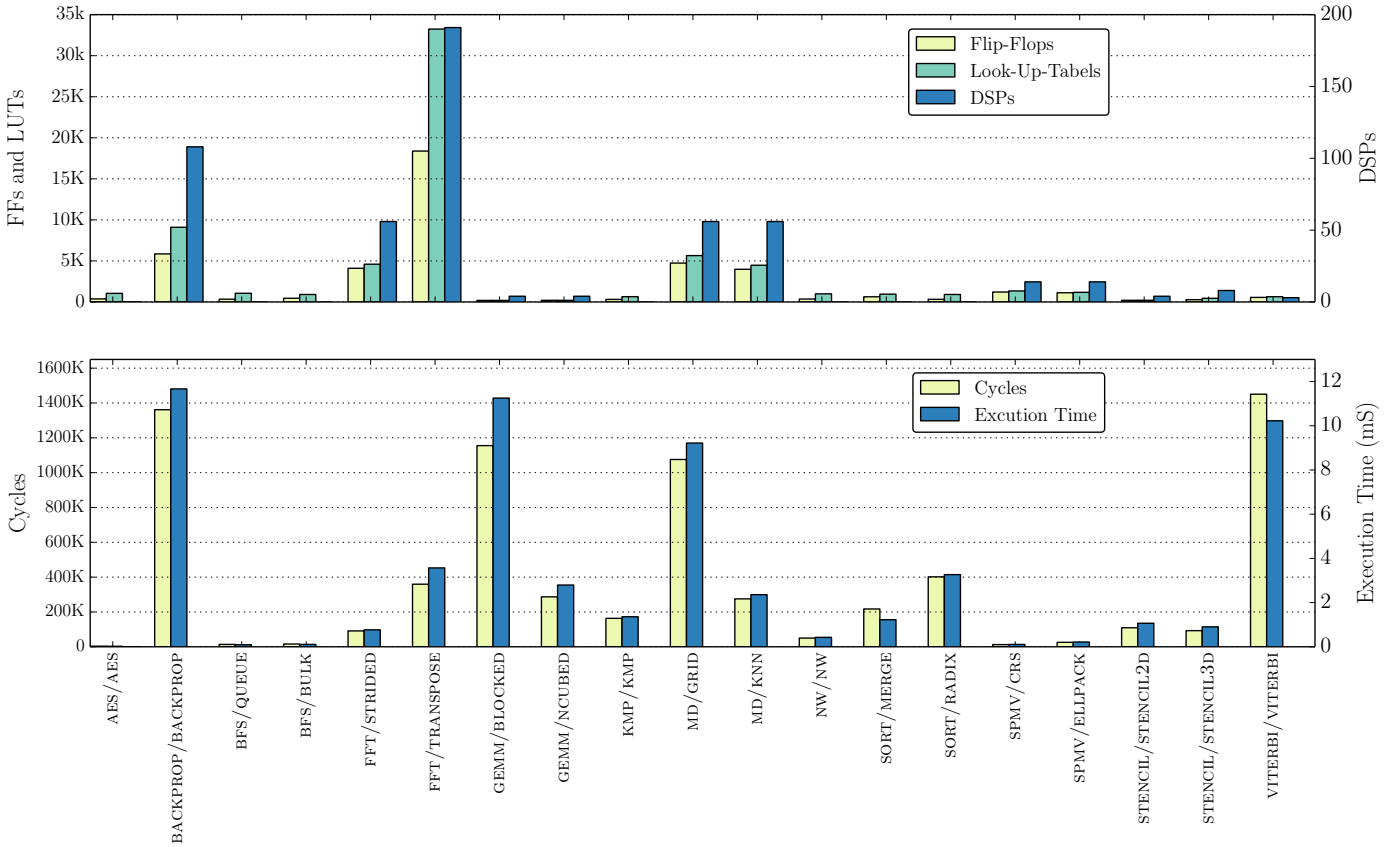


Fig. 7: MachSuite Synthesis results.

benchmarks is immediately obvious. The relationships between GEMM/BLOCKED and GEMM/NCUBED is more subtle. GEMM/BLOCKED takes longer to execute than its naive, triple nested loop counterpart. This is an artifact of HLS.

The blocked version is more complex (with more nested loops) and requires directives to be efficient. Additionally, blocking is meant to increase locality and reduce memory contention, increasing parallelism and reducing memory costs. Without any directives for pipelining or parallelization, the optimizations that GEMM/BLOCKED is meant to exploit never materialize. Only one SRAM is allocated, so GEMM/BLOCKED ends up paying the price of its code structure without reaping the rewards. This type of experimentation with HLS tools, optimization, and hardware techniques is exactly the kind of scenario that MachSuite is meant to support.

VI. Conclusion

MachSuite provides a clean, HLS synthesizable code base to enable standardization and commensurability in accelerator-centric research. It lessens the burden of manually implementing hardware accelerators with ad-hoc benchmark selection and smooths the way for architecture researchers to access a diverse set of hardware accelerators. Moreover, the diverse characteristics of the benchmarks in MachSuite pose challenges to explore customization strategies in memory hierarchy and system integration. MachSuite also supplies the HLS community with a common ground on which to compare results; HLS tool developers can use MachSuite to exercise their new optimizations to handle more complex program behaviors and architects can use MachSuite to stress test their proposed solutions in a more rigorous, scientific manner.

VII. Acknowledgments

We would like to thank Glenn Holloway and Yu Wang for their help revising this work. This work was partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA and the National Science Foundation (NSF) Expeditions in Computing Award #: IIS-0926148. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] The harwell-boeing collection. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/.

[2] Lammps molecular dynamics simulator. http://lammps.sandia.gov/.

[3] Scalable molecular dynamics. http://www.ks.uiuc.edu/Research/namd/.

[4] M. Alle, A. Morvan, and S. Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, May 2013.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[6] J. Auerbach, D. Bacon, P. Cheng, R. Rabbah, and S. Shukla. Virtualization of heterogeneous machines. In *(DAC)*, pages 890–894, June 2011.

[7] A. Canis, J. H. Anderson, and S. D. Brown. Multi-pumping for resource reduction in fpga high-level synthesis. In *(DATE)*, pages 194–197, March 2013.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *CS Department, Carnegie Mellon University*, 2004.

[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 *(IISWC)*, 2009.

[10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads. In *(IISWC)*, 2010.

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[12] E. S. Chung, J. D. Davis, and J. Lee. Linqits: big data on little clients. *ISCA*, 2013.

[13] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO*, 2010.

[14] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *(ISLPED)*, pages 305–310, Sept 2013.

[15] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. Charm: a composable heterogeneous accelerator-rich microprocessor. In *ISLPED*, 2012.

[16] J. Cong, K. Gururaj, and G. Han. Synthesis of reconfigurable high-performance multicore systems. In *FPGA*, 2009.

[17] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou. Combining module selection and replication for throughput-driven streaming programs. DATE '12, pages 1018–1023, San Jose, CA, USA, 2012. EDA Consortium.

[18] 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*, 2010.

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008.

[20] A. Del Barrio, M. Molina, J. Mendias, R. Hermida, and S. Memik. Using speculative functional units in high level synthesis. In *(DATE)*, pages 1779–1784, March 2010.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[22] K. Fan, M. Kudlur, G. S. Dasika, and S. A. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA*, 2009.

[23] K. Han, K. Choi, and J. Lee. Compiling control-intensive loops for cgras with state-based full predication. In *(DATE)*, pages 1579–1582, March 2013.

[24] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *ISCAS*, pages 1192–1195. IEEE, 2008.

[25] P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, 2007.

[26] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *PACT*, 2011.

[27] D. E. Knuth, J. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[28] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[29] I. Levin. A byte-oriented aes-256 imlpementation. www.literatecode.com, 2009.

[30] S. Li. Sparse matrix techniques (tutorial). 2007.

[31] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August. Cgpa:coarse-grained pipelined accelerators. In *DAC*, 2014.

[32] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *DAC*, 2013.

[33] H.-Y. Liu, M. Petracca, and L. P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *DATE*, 2012.

[34] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *TACO*, 2012.

[35] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza. A high-level synthesis flow for the implementation of iterative stencil loop algorithms on fpga devices. DAC '13, pages 52:1–52:6, New York, NY, USA, 2013. ACM.

[36] S. Neuendorffer and F. Martinez-Vallina. Building zynq accelerators with vivado high level synthesis. In *FPGA*, 2013.

[37] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[38] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. ISCA '07, pages 412–423, New York, NY, USA, 2007. ACM.

[39] F. I. P. S. Publications. Announcing the advanced encryption standard. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[40] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ISCA*, 2013.

[41] B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks. Quantifying acceleration: Power/performance trade-offs of application kernels in hardware. In *ISLPED*, 2013.

[42] Y. S. Shao and D. Brooks. Isa-independent workload characterization and its implications for specialized architectures. In *ISPASS*, 2013.

[43] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA*, 2014.

[44] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Number IMPACT-12-01, Urbana, Mar. 2012.

[45] F. Vazquez, E. M. Garzon, J. A. Martnez, and J. J. Fernandez. The sparse matrix vector product on gpus. In *NVidia Tech Reports*, 2009.

[46] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *MICRO*, 2011.

[47] V. Volkov and B. Kazian. Fitting fft onto the g80 architecture. 2008.

[48] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. DAC '13, pages 12:1–12:8, New York, NY, USA, 2013. ACM.

[49] S. Weerawarana, E. N. Houstis, and J. R. Rice. An interactive symbolic-numeric interface to parallel ellpack for building general pde solvers. In *Tech Reports, Purdue University*, 1990.

[50] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *SC*, 2005.

[51] F. Winterstein, S. Bayliss, and G. Constantinides. High-level synthesis of dynamic data structures: A case study using vivado hls. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 362–365, Dec 2013.

[52] T. Yokota, K. Ootsu, and T. Baba. Introducing entropies for representing program behavior and branch predictor performance. In *Workshop on Experimental Computer Science*, 2007.