Virtual Instruction Set Computing for Heterogeneous Systems *

Vikram Adve, Sarita Adve, Rakesh Komuravelli, Matthew D. Sinclair and Prakalp Srivastava
University of Illinois at Urbana-Champaign
{vadve,sadve,komurav1,mdsincl2,psrivas2}@illinois.edu

Abstract

Developing software applications for emerging and future heterogeneous systems with diverse combinations of hardware is significantly harder than for homogeneous multicore systems. In this paper, we identify three root causes that underlie the programming challenges: (1) diverse parallelism models; (2) diverse memory architectures; and (3) diverse hardware instruction set semantics. We believe that these issues must be addressed using a language-neutral, virtual instruction set layer that abstracts away most of the low-level details of hardware, an approach we call Virtual Instruction Set Computing. Most importantly, the virtual instruction set must abstract away and unify the diverse forms of parallelism and memory architectures using only one or two models of parallelism. We discuss how this approach can solve the root causes of the programmability challenges, illustrate the design with an example, and discuss the research challenges that arise in realizing this vision.

1 Introduction

The future of computing is heterogeneous. Single chips currently exist with several billion transistors, and this number will continue to increase for at least the next decade [9]. However, power dissipation in these chips is an increasing problem, especially given the limited power envelopes of battery-powered devices. Given this, we have two options: turn off large portions of the chip (the problem known as Dark Silicon [20]) or find more power efficient options to keep the transistors on.

Heterogeneous computing falls into the second category. It provides the ability to integrate a variety of processing elements, such as general-purpose cores, GPUs, DSPs, FPGAs, and custom or semi-custom hardware into a single system. If applications can execute code on the device which best suits it, then heterogeneous systems can provide higher energy efficiency than conventional processors. In the best case, customized hardware accelerators have been shown to provide 100x-1000x better power efficiency for specific computations [23, 26].

*This work was funded in part by NSF Award Numbers 0720772 and CCF-1018796, and by the Intel-sponsored Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign.

However, there are numerous challenges to getting such a system to operate effectively and efficiently. One major challenge is the difficulty of programming applications to use diverse computing elements. We identify three fundamental root causes that underlie these challenges: (1) diverse models of parallelism; (2) diverse memory architectures; and (3) diverse hardware instruction sets and execution semantics. We discuss these root causes and the challenges they engender in Section 2.

In this paper, we describe a broad vision and some preliminary design choices for solving the programmability problem by eliminating all these three root causes. We believe that this can be achieved only by abstracting away the differences in heterogeneous hardware, and presenting a more uniform hardware abstraction across devices to software. More specifically, using a low-level, language-neutral, virtual instruction set can encapsulate all the relevant programmable hardware components on target systems. In this instruction set, source-level applications are compiled, optimized, and shipped as "virtual object code" and then translated down to a specific hardware configuration, usually at install time, using *system-specific* compiler back ends ("translators"). We call this strategy *Virtual Instruction Set Computing* (or VISC, as opposed to CISC or RISC) [2].

This broad strategy is not new – it has been used in a few commercial systems such as IBM System/38 and AS/400, Transmeta processors, NVIDIA's PTX and Microsoft's DirectCompute [16, 36, 14, 18, 32, 1] and explored in a few research projects [19, 35, 2]. PTX and DirectCompute have used this approach very successfully to abstract away families of GPUs and are strong evidence that the VISC approach is commercially viable and can deliver high performance. Addressing a wider range of heterogeneous hardware, however, requires solving all the three root causes above. PTX and DirectCompute only partially address the challenges of diverse memory architectures and of diverse hardware instruction sets, focusing only on the case of GPUs. We discuss further details of these and other current efforts in Section 3.

The key novelty in our work is that our instruction set exposes a very small number of models of parallelism and a few memory abstractions essential for high per-

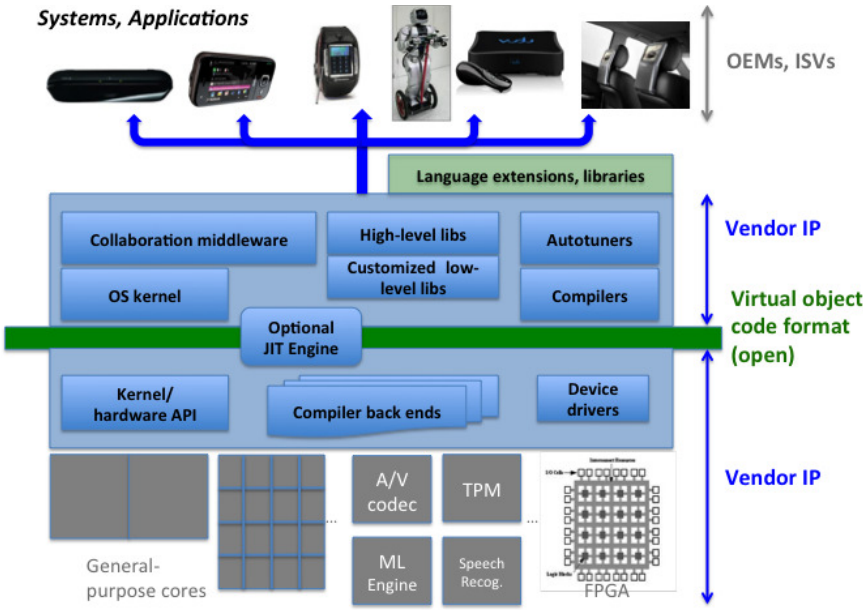

Figure 1: System Organization for Virtual Instruction Set Computing in a Heterogeneous System

formance algorithm development and tuning. Together, we expect that these abstractions will effectively capture a wide range of heterogeneous hardware, which would greatly simplify major programming challenges such as algorithm design, source level portability, and performance tuning. Our overall approach is illustrated in Figure 1 and is discussed further in Section 4. We conclude by discussing open research problems in Section 5.

2 Programmability Challenges

Heterogeneous parallel computing systems, including both mobile System-on-Chip (SOC) designs such as Qualcomm's Snapdragon and nVidia's Tesla, or high-end supercomputers like Cray's BlueWaters (which has many GPU coprocessors) or Convey's FPGA-based HC-1, raise numerous difficult programming challenges. We believe these challenges arise from *three fundamental root causes* and we first discuss these root causes and then outline the challenges they engender.

Root Causes of Programmability Challenges:

(1) Diverse models of parallelism: Different hardware components in heterogeneous systems support different models of parallelism. We tentatively identify five broad classes of *programmable* hardware that have qualitatively different models of parallelism:

1.	General purpose cores	Flexible multithreading
2.	Vector hardware	Vector parallelism
3.	GPUs	Restrictive data parallelism
4.	FPGAs	Customized dataflow
5.	Custom accelerators	Various forms

In addition, applications running on multiple such components may exhibit asynchronous or synchronous parallelism relative to each other.

(2) Diverse memory architectures: With the different parallel models come deep differences in the memory system. Common choices in the various components above include cache-coherent memory hierarchies, vector register files, private or "scratchpad" memory, stream buffers, and custom memory designs used in custom accelerators. These differences in memory architectures strongly influence both algorithm design and application programming. Moreover, the performance tradeoffs are becoming even more complex as new architectures provide more options, e.g., nVidia's Fermi architecture allows a 64 KB block of SRAM to be partitioned flexibly into part L1 cache and part private scratchpad memory.

(3) Diverse hardware-level instruction sets and execution semantics: Finally, the various hardware components have very different instruction sets, register architectures, performance characteristics, and execution semantics. These differences have an especially profound effect on object-code portability. They also have other negative effects, described below.

Major Programmability Challenges:

These fundamental forms of diversity create deep programmability challenges for heterogeneous systems. First, it is extremely difficult *to design a single algorithm* for a given problem that works well across a range of such different models of parallelism, with such different memory systems, as previous work has shown [33, 4]. We envisage two options to address this problem: design

algorithms that achieve good, but not optimal, performance across the targeted range of hardware, or use multiple algorithms for a given problem and select among them when the actual hardware configuration is known (e.g., at install time, load time, or run time) [28, 4]. In practice, both approaches will likely be necessary.

Second, it is much more difficult *to design effective source-level programming languages* for heterogeneous systems. A single programming language or library typically supports only one or two models of parallelism. For example, CUDA, OpenCL and AMP naturally support fine-grain data parallelism, with a macro function replicated across a large number of threads, but other parallelism models (like more flexible dataflow) are not specifically addressed. Similarly, BlueSpec [30] and Lime [5], which have proved successful for FPGAs, are both dataflow models but it is not clear whether these can be mapped effectively to GPUs. The consequence today is that, a programmer must program each hardware component differently, which creates a huge barrier to entry for widespread use of heterogeneous hardware.

A third challenge is *source code portability*. Heterogeneous systems can provide *different combinations of hardware*, both within a single manufacturer's family of devices and across different manufacturers' devices. This makes source-code level portability difficult, in two ways. First, each component must solve the algorithm portability problem (above). Second, compilers must map source code embodying one or more algorithms for each component down to the various hardware components on which those algorithms must run. This task is greatly complicated by all three forms of diversity.

Fourth, *performance tuning* for heterogeneous systems will also be significantly more complex. The disparate parallelism models, memory architectures, and lower-level performance details require significantly different performance models and tuning strategies. Because of these disparities, the programmer training, software tools, and application libraries all become prohibitively expensive as the number of different hardware components grows.

Finally, *object-code portability* across the same and different manufacturers' devices is essential as well. An application vendor must be able to ship a single software version for a broad range of devices – it is impractical to create, test, market and support different versions of an application package for all the different devices running a single platform, e.g., all the smartphones running Android. Today, Android solves this problem by using Java bytecode, *but only for CPU application code: few application components take advantage of the on-phone GPU or DSPs*, and those are usually native libraries. Moreover, the debuggers, profilers and performance tools for a family of heterogeneous systems must support the full range of available hardware, both within a single system and also across different system configurations. Because both tuning and debugging often need to go down to the level of object code, these tools become expensive to develop, learn and use for each family of hardware.

3 State of the Art

Most of the existing research on programming heterogeneous systems has focused on source level programming models and languages. Existing languages like CUDA, OpenCL, AMP and OpenACC are primarily focused on GPU computing (The OpenMP standards committee is developing OpenMP extensions for accelerators, which are expected to be very similar to OpenACC [6].) In particular, they primarily support a single parallelism model: parallel execution of a kernel function replicated across a large number of cores, with explicit copying of data from host to device and back. (OpenCL supports task parallelism, but that model is a poor match to GPUs, which are the primary targets of existing OpenCL implementations. For example, it has been shown that CUDA programs that aren't data parallel often perform poorly on GPUs [34].) All commercial implementations we know of focus on GPUs and general purpose multicore CPUs and do not address other components in a heterogeneous system.

The Liquid Metal project [24] aims to program hybrid CPU/accelerator code, using a single object-oriented programming language called Lime. Their efforts to date have focused on CPU/FPGA systems. The FPGA part is compiled first to a language based on a dataflow graph of filters connected by input/output queues, which in turn is compiled to Verilog. The approach is very specific to FPGAs and it is not clear if it would be effective for more diverse components.

Delite [11] takes a promising approach. Broadly, Delite provides a framework to create implicitly parallel domain-specific languages (DSLs) and a runtime to execute these parallel DSLs on a heterogeneous platform. The Delite run-time creates a dynamic execution graph of method executions along with their dependencies, and this graph is then scheduled across the system. Because DSLs can be flexible and high-level, the DSL approach could be used to shield the programmer from most of the differences between different heterogeneous devices. In practice, however, this puts enormous burden on the compiler and run-time system to translate the high-level language to a variety of different hardware components to achieve adequate performance. For example, in Delite, regular data-parallel kernels are automatically translated to the target device language (e.g., CUDA for GPU), but for irregular ones, the DSL author is required to provide a hand-written CUDA kernel, which undercuts the promise of high-level programming.

At the object code level, the NVIDIA's PTX and

Microsoft's DirectCompute are virtual instruction sets for GPU computing. Both these systems aim to support a stable programming model and instruction set (for CUDA and AMP programs respectively), while also providing efficient code. These instruction sets expose a data-parallel programming model and provide object code portability, PTX across NVIDIA GPUs but DirectCompute across a wide range of GPUs. Their primary limitation is that they do not support other classes of hardware. In particular, they do not aim to address the first root cause we have identified – expose only a restrictive throughput-oriented data parallelism and do not expose other models of parallelism like dataflow-style parallelism, or general task parallelism required across different compute elements. Also PTX and DirectCompute only partially address the second root cause – abstract away memory architectures only within a single class of heterogeneous components, GPUs and thus not supporting other classes of heterogeneous components. Nevertheless, they do provide strong evidence that a virtual instruction set approach is commercially viable, highly scalable, and can be used to achieve high performance.

4 The VISC Approach

4.1 Basics of The VISC Design Strategy

We propose to leverage Virtual Instruction Set Computing (VISC) to address all three root causes of the programmability challenges for heterogeneous systems. Our system organization is shown in Figure 1. The key point in the figure is that the only software components that can "see" the hardware details in a system are the translators (i.e., the compiler back ends), a minimal set of low-level OS components sufficient to implement a full-featured OS kernel [17], and potentially some device drivers. The rest of the software stack, including most of an OS kernel, higher-level language compilers and other development tools, application-independent libraries, frameworks and middleware, and all application software, live above the virtual ISA.

To our knowledge, the VISC approach has never before been applied to multiple kinds of heterogeneous hardware, which presents unique and difficult challenges. To solve the three root causes of programmability problems described earlier, the VISC approach must have three features: (i) A very small number of fundamental abstractions for parallel computation (preferably, only one or two) in the virtual instruction set; (ii) abstractions for memory and communication that support algorithm design, source level language compilation, and application performance tuning; and (iii) a uniform instruction set that can be mapped down to different hardware instruction sets and execution models in different heterogeneous hardware components. We discuss the first two briefly, in turn. The third feature we expect to inherit directly by building our virtual ISA on top of the LLVM instruction set, which has already proven extremely effective at abstracting away a wide range of sequential hardware instructions sets [2, 27, 29].

4.2 Abstractions for Individual Elements

Because the Virtual ISA is an abstraction of the hardware, its design must be driven by the classes of hardware it represents. It is important that the virtual ISA not simply be a union of 5 different kinds of abstractions for the five broad classes of hardware identified in Section 2.

We expect that virtually all the parallelism exploited in *individual* non-CPU elements will be data parallelism in the broadest sense, including both regular and irregular data parallelism. Therefore, we focus on mapping data parallelism to individual computing elements, and leave more general functional or task parallelism to be handled by parallel execution across elements, discussed together with memory abstractions, below.

In designing the virtual ISA for individual elements, we have considered several parallelism models that have been used for different hardware elements today, as well as combinations of them that can be integrated cleanly. In the interests of space, we omit the discussions of our analysis and describe the tentative outcome instead. All these choices will build on some variant of the sequential LLVM instruction set.

Our tentative strategy is to use a combination of two parallelism abstractions: vector parallelism and generalized macro dataflow graphs. A virtual vector instruction set, such as Vector LLVA [8] or Vapor SIMD [31], is an obvious candidate because vectorizable code is likely to capture a large category of codes (though not all) that run successfully on GPUs, FPGAs, and vector hardware. Moreover, when applicable, vector instructions are simple, have intuitive deterministic semantics, and can be made highly portable through careful design of the vector lengths, control flow, memory alignment, and operation abstractions [8, 31].

For computations not cleanly expressible as vector code, our second choice is a general dataflow graph, which may include fine-grain dataflow such as Static Single Assignment (SSA) form within procedures (which already exists in LLVM) and "macro dataflow" representations for coarse-grain data flow. This dataflow graph will not be "pure dataflow" in the sense that individual nodes may include side-effects, i.e,. reads and writes, to shared memory, because explicitly managing memory accesses may be important for performance tuning and front-end compiler optimization. Although these side effects may cause concurrency errors like data races or unintentional non-determinism, it is important to note that the virtual ISA is *not a source-level programming language* but an object code language: guaranteeing correctness properties like the absence of concurrency errors is not the re-

sponsibility of the virtual ISA and is instead left to source level languages.

We believe that such a (less restrictive) dataflow graph language will be able to capture all the forms of data parallelism. For example, the implicit "grid of kernel functions" used in PTX [32], Renderscript [3], WebGL [37], and DirectCompute [1] are naturally expressible as dataflow graphs. While dataflow graphs can also capture vectorizable code, they may be less efficient and less compact than vector instructions, which is why we expect to use vector instructions as well.

Moreover, having one monolithic dataflow graph for an entire application would severely restrict the modularity and flexibility of the code. We therefore propose to make the dataflow graphs hierarchical by simply allowing the individual nodes in the overall graph to be either a vector subprogram, or another dataflow graph of its own. Individual nodes in the graph would represent computation within each element, either as simple vector code or rich fine-grain dataflow graphs. Different nodes can use different choices for different kernels. Together this combination would give an elegant, yet powerful, mechanism for expressing parallelism across an entire heterogeneous system since a wide range of high-level data-parallel programming models can be compiled efficiently down to either a dataflow model or a vector model, including OpenCL, Array Building Blocks [21], CUDA, Renderscript, OpenMP, Fortran 9x, Hierarchical Tiled Arrays (HTAs) [7, 22], Concurrent Collections [10], Lime, StreamIt [38], and others.

In Listing 1, we give an example of the hierarchical dataflow representation of the FFT radix-2 algorithm, which is the simplest and most commonly used form of the Cooley and Tukey FFT algorithm [15], in our proposed macro dataflow language. We have studied two implementations of this algorithm, an FPGA implementation from the ERCBench benchmark suite [12], and a GPU implementation from the Parboil benchmark suite [25] and tried to capture them in a common dataflow representation.

Figure 2 gives the pictorial view of the dataflow graph created in Listing 1. *FFT* is a node in the graph, which in turn consists of $\log_2 N$ *Stage* nodes, where each *Stage* node further consists of $N/2$ *Butterfly* nodes. In *main()* we first describe the node hierarchy in the dataflow graph. Next we use the *generate_edges()* function to create the dataflow edge connections among different nodes in the graph. As shown in the figure we have classified these edges into three types ($E_{stage-stage}$, $E_{butterfly-stage}$ and $E_{stage-fft}$) based on the type of nodes they connect. Additionally, these edges are capable of carrying user-defined data types such as *Complex*. The example shows that the dataflow graph is capable of exposing the parallelism exploited by the GPU and

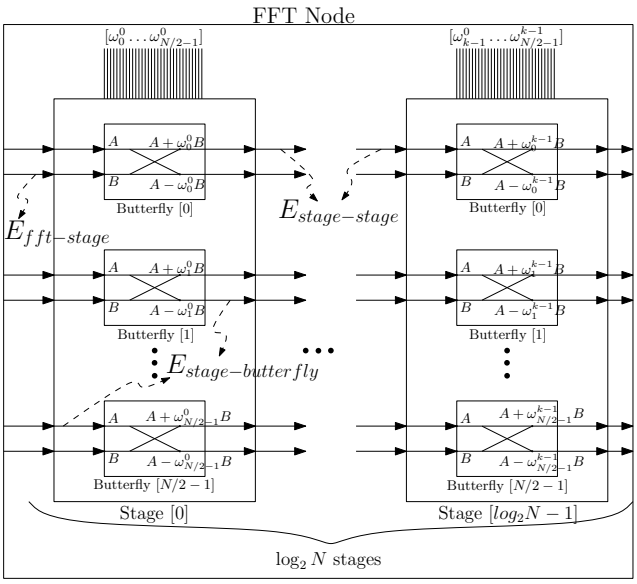

Figure 2: Macro Dataflow representation of FFT
E_{X-Y} denotes dataflow edges between nodes X and Y.

FPGA implementations. The utility of abstracting away the low level details of data movement through dataflow edges is given in Section 4.3.

4.3 Memory System Abstractions

An important aspect of VISC is what abstraction of the overall memory system we present to programmers for coordinating parallelism across compute elements. When programming GPUs in CUDA or OpenCL, we need to explicitly manage data movement between the CPU and the GPU, which is an additional overhead to the programmer. Additionally, newer accelerators like GPUs based on NVIDIA's Fermi architecture combine increasingly flexible options for caches or scratchpad memory. (In an allied research project, we are exploring much more flexible memory system designs, combining highly reconfigurable SRAM memory blocks for energy efficiency and performance.) Unfortunately, these kinds of designs enormously complicate the task of application developers, who have to optimize their code to take advantage of such hardware features. Our VISC design approach aims to enable such flexibility for hardware designers, without putting an excessive burden on application designers. Applications should be able to express an overall computation in a simple and abstract computational model. Such a model would convey the essential coordination and data transfer requirements and leave the low-level mapping and execution details to the underlying compiler, scheduler and hardware.

The hierarchical dataflow graphs proposed in Section 4.2 provide a clean, flexible communication and coordination abstraction across computing elements. For

```
//————————Global declarations————————
struct Complex = type {float real, float imag};
const int k = log2 N;
Node FFT (Complex In[]) : (Complex Out[]);
Node Stage (Complex In[], Complex ω[])
                        : (Complex Out[]);
Node Butterfly(Complex A, Complex B,Complex ω)
                    : (Complex O1, Complex O2);
//————————————Functions————————————
Complex complexAdd(Complex A, Complex B) {
  // complexSub(), complexMul() are similar
  Complex result;
  result.real = A.real + B.real;
  result.imag = A.imag + B.imag;
  return result;
}
(Complex, Complex) computeButterfly (Complex A,
     Complex B, Complex ω) {
  Complex temp = complexMul(B, ω);
  Complex O1 = complexAdd(A, temp); // A + Bω
  Complex O2 = complexSub(B, temp); // A − Bω
  return (O1, O2);
}
main() {
  generate_nodes();
  generate_dataflow_edges();
}
//————————Generate node hierarchy————————
generate_nodes() {
  Butterfly := Node 1 of computeButterfly();
  Stage     := Node N/2 of Butterfly;
  FFT       := Node k of Stage;
}
//————————Generate dataflow edges————————
generate_dataflow_edges() {
  // E_stage−stage: Inter−stage edges
  for i in 0 to k − 2 do
    for j in 0 to N/2 − 1 do
      expr = ⌊j/2^i⌋ * 2^(i+1) + j  mod  2^i;
      Stage[i].Out[2j] => Stage[i+1].In[expr];
      Stage[i].Out[2j+1] => Stage[i+1].In[expr
          +2^i];
  // E_butterfly−stage: Edges between a stage and
  // its butterfly nodes
  for i in 0 to k − 1 do
    for j in 0 to N/2 − 1 do
      with(Stage[i]) {
        In[j] => Butterfly[j].A;
        In[j + N/2] => Butterfly[j].B;
        ω[j] => Butterfly[j].ω;
        Butterfly[j].O1 => Out[2j];
        Butterfly[j].O2 => Out[2j+1];
      }
  // E_stage−fft : Edges between FFT, stage nodes
  for m in 0 to N − 1 do
    FFT.In[m] => Stage[0].In[m];
    Stage[k−1].Out[m] => FFT.Out[m];
}
```

Listing 1: Example Dataflow Representation for FFT. '=>' denotes an edge; ':=' denotes a node definition.

instance, in Listing 1 we abstract away the data movement details in the FPGA and GPU (e.g. CUDA memcpy operations) through edges which move data from the producer node to the consumer node. This dataflow across elements can then be efficiently mapped to either uncached local memories or to cache-coherent shared memory. Although explicit loads and stores of the GPU and FPGA implementations have been abstracted away in this example, we do expect to need explicit loads/-stores, which is not uncommon in macro dataflow languages. For example, an alternative approach to expressing the FFT algorithm would be to use loads and stores inside the Stage node for transferring data between successive stages, instead of using explicit dataflow edges as shown. (We omit that pseudocode for lack of space.)

5 Open Research Problems

We must solve several key problems to bring these ideas to fruition. We must design a back-end compilation process from the Virtual ISA to native code for each hardware device, which can exploit Virtual ISA information to perform hardware dependent optimizations. We must develop compilation techniques that translate communication and data movement down to hardware with a variety of memory architectures, e.g., some combination of non-cached local memories, cache-coherent shared memory, or streaming memory. We must design an autotuning partitioner that decides how to partition the program effectively between diverse hardware elements, tune individual component code for specific hardware, and guide the run-time scheduler. Finally, we must design run time schedulers that take advantage of information extracted by autotuning ahead of time and the flexibility offered by the Virtual ISA to optimize power and execution time.

Another key direction of work is more effective, software-aware memory organization. To minimize energy usage or to maximize performance in a heterogeneous system, depending on the computation, a given set of data may ideally be stored as hardware-managed cache lines, software-managed scratchpads with flexible data granularities and layouts, arbitrary messages, vector registers, etc. Moreover, hardware caching today is largely software-oblivious but there is much information in software to optimize how memory is managed. In prior work on the DeNovo architecture, we have successfully used program-level information about data sharing and access for efficiencies in power, performance, and complexity when managing data for homogeneous multicores with caches [13]. These ideas can be extended for even greater benefits in heterogeneous systems. To accomplish these gains, the Virtual ISA must provide a general abstraction to exploit all such storage structures, the backend compiler must map this abstraction efficiently, and the hardware must be designed to exploit this flexibility.

References

[1] DirectCompute Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf, 2010.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003.

[3] Android Developers. Renderscript. http://developer.android.com/reference/android/renderscript/package-summary.html.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, 2009.

[5] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, 2010.

[6] J. Beyer, E. Stotzer, A. Hart, and B. D. Supinski. OpenMP for Accelerators. 2011.

[7] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, 2006.

[8] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, 2006.

[9] J. Brodman, B. Fraguela, M. J. Garzaran, and D. Padua. Design Issues in Parallel Array Languages for Shared Memory. In *8th Int. Workshop on Systems, Architectures, Modeling, and Simulation*, 2008.

[10] Z. Budimlic, M. Burke, V. Cav, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent Collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[11] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, 2011.

[12] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, et al. ERCBench: An open-source benchmark suite for embedded and reconfigurable computing. In *International Conference on Field Programmable Logic and Applications*, FPL, 2010.

[13] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *20th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2011.

[14] B. E. Clark and M. J. Corrigan. Application System/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423, 1989.

[15] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput*, 19(90):297–301, 1965.

[16] I. Corporation. System/38-A high-level machine. *IBM SYSTEM/38 Technical Developments*, 1978. available through IBM branch offices.

[17] J. Criswell, B. Monroe, and V. Adve. A Virtual Instruction Set Interface for Operating System Kernels. In *WIOSCA*, 2006.

[18] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *CGO*, 2003.

[19] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. Int'l Conf. on Computer Architecture (ISCA)*, pages 26–37, 1997.

[20] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA, 2011.

[21] A. Ghuloum, A. Sharp, N. Clemons, S. D. Toit, R. Malladi, M. Gangadhar, M. McCool, and H. Pabst. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures. http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=227300084, 2010.

[22] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, 2008.

[23] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA, 2010.

[24] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP, 2008.

[25] IMPACT Research Group and others. Parboil benchmark suite, 2007.

[26] R. Iyer, S. Srinivasan, O. Tickoo, Z. Fang, R. Illikkal, S. Zhang, V. Chadha, P. M. Stillwell, and S. E. Lee. CogniServe: Heterogeneous Server Architecture for Large-Scale Recognition. *IEEE Micro*, 31(3):20–31, 2011.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[28] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[29] LLVM User Community. `http://llvm.org/Users.html`.

[30] R. S. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings of Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE, 2004.

[31] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2011.

[32] Nvidia Compute. PTX: Parallel Thread Execution ISA Version 2.3. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf`, 2011.

[33] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment. In *Proceedings of the 7th international conference on High performance computing for computational science*, VECPAR, 2007.

[34] M. D. Sinclair. Enabling New Uses for GPUs. Master's thesis, University of Wisconsin-Madison, 2011.

[35] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving High Performance via Co-designed Virtual Machines. In *Proc. Int'l Workshop on Innovative Architecture*, IWIA, 1999.

[36] F. Soltis. Design of a small business data processing system. *IEEE Computer*, 14:77–93, 1981.

[37] The Khronos Group. WebGL - OpenGL ES 2.0 for the Web. `http://www.khronos.org/webgl/`, 2009.

[38] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, 2002.