Computing without Processors

Heterogeneous systems allow us to target our programming to the appropriate environment

Satnam Singh, Microsoft Research Cambridge, UK

From the programmer's perspective the distinction between hardware and software is being blurred. As programmers struggle to meet the performance requirements of today's systems, they will face an ever increasing need to exploit alternative computing elements such as GPUs (graphics processing units), which are graphics cards subverted for data-parallel computing,¹¹ and FPGAs (field-programmable gate arrays), or soft hardware.

The current iteration of mainstream computing architectures is based on cache-coherent multicore processors. Variations on this theme include Intel's experimental Single-Chip Cloud Computer, which contains 48 cores that are not cache coherent. This path, however, is dictated by the end of frequency scaling rather than being driven by requirements about how programmers wish to write software.⁴ The conventional weapons available for writing concurrent and parallel software for such multicore systems are largely based on abstractions developed for writing operating systems (e.g., locks and monitors). However, these are not the right abstractions to use for writing parallel applications.

There are better ways to bake all that sand. Rather than composing many elements that look like regular CPUs, a better approach, from a latency and energy-consumption perspective, is to use a diverse collection of processing elements working in concert and tuned to perform different types of computation and communication. Large coarse-grain tasks are suitable for implementation on multicore processors. Thousands of fine-grain data-parallel computations are suitable for implementation on GPUs. Irregular fine-grain tasks that need to run with extreme requirements for performance or reduced energy consumption are suitable for implementation as digital circuits running on an FPGA chip. The future is heterogeneous.

The emergence of such heterogeneous systems challenges the comfortable distinction that has been established between the design processes for hardware and software. The ISA (instruction set architecture) of the CPU has acted as a convenient interface layer. The software industry was predicated on the assumption that there is only one kind of hardware (the CPU) with a standardized interface (the ISA). This allowed the hardware industry continually to innovate a series of microarchitectures that conformed to the fixed ISA interface. This cozy demarcation is now being eroded as programmers seeking the absolute maximum performance or reduced energy consumption resort to alternative processing elements such as GPUs and FPGAs.

One place we are likely to see the deployment of heterogeneous computing systems is in the cloud where we can imagine racks populated with not only multicore processors but also GPUs and FPGAs. Amazon has already created the Elastic Compute Cloud, which allows computations to be executed on GPUs. Some kinds of computations can be executed on GPUs and FPGAs at a performance-per-dollar ratio that is significantly better than what is achievable with a CPU. As energy use becomes more of a limiting factor in the growth of data centers, the deployment of heterogeneous architectures to help reduce latency and energy consumption will be inevitable. Such

systems will need programming models and runtime infrastructures that allow the remapping of computations to different types of processing hardware, depending on the computational demands and nonfunctional requirements (energy versus latency). These systems will even open up the opportunity to charge based on energy consumption rather than time or cycles.

To make heterogeneous computing in the cloud a reality there are still many technical challenges to overcome: for example, how to virtualize GPU and FPGA computations so they can be moved to different physical resources and how to guard against security violations.⁹

Another challenge introduced by heterogeneous systems will be the design of data structures that are suited for use on multicore processors and heterogeneous computing elements connected with a complex memory hierarchy or on-chip network. This will force us to rethink structures such as stacks and queues and instead consider unordered concurrent constructs that allow greater flexibility for memory access and layout.¹²

WHAT IS POSSIBLE TODAY

Modern heterogeneous systems are evolving toward a future of intriguing computing possibilities. Today's systems are in various stages of development in achieving that goal, including the refinement of architectures that allow parallel processing, innovations to existing programming languages, and the expansion of language portability to GPUs, AVX (Intel's Advanced Vector Instructions), and FPGAs.

MORE THAN NESTED PARALLEL FOR LOOPS

The most popular heterogeneous computing resource today is the GPU, which provides an architecture with a high aggregate memory bandwidth and the ability to perform many more data-parallel operations than is possible with a conventional processor. Other heterogeneous computing elements such as FPGAs can also efficiently realize data-parallel operations, as well as supporting high-speed irregular operations such as processing XML queries at "line speed" (i.e., in realtime as data flows over a high-speed network).

The exploitation of new data-parallel computing elements has naturally drawn on the experience of parallel programming in high-performance scientific computing (e.g., MPI and OpenMP). Many useful applications are not data parallel yet do permit other kinds of parallelization (e.g., asynchronous communicating threads where each performs a different task).

To make effective use of heterogeneous computing, programmers need to look beyond the problem of how to parallelize nested for loops and consider how to compute as much as possible in parallel in the highly irregular structure of modern applications. Techniques that make it easier to exploit irregular parallelism include STM (software transactional memory) and AME (automatic mutual exclusion).¹

DATA-PARALLEL COMPUTING WITH GPUS

Although GPUs can offer substantial performance improvements for data-parallel algorithms, they are far from trivial to program even with special GPU programming languages and systems such as Nvidia's CUDA (Compute Unified Device Architecture).¹⁰ To get good performance from GPUs, one undergoes a "Back to the Future" experience where tomorrow's silicon is programmed using yesterday's techniques—that is, explicit data placement, data movement, and data synchronization.

Modern GPU architectures have a memory hierarchy that needs to be explicitly programmed to obtain good performance.

Today most people who make effective use of GPUs undergo a steep learning curve and are forced to program close to the machine using special GPU programming languages. Do programmers really need a special set of languages to write data-parallel programs for GPUs and a different set of languages to write data-parallel programs for multicore processors? At a high level of abstraction the task at hand is the same in both scenarios.

The software industry is reacting to the emergence of multicore systems by adding innovations to existing languages to help express concurrent and parallel computing. The use of data-parallel computing on GPUs, however, could not wait for languages such as C++ to be modified to support parallelism better, so systems such as CUDA were developed to fill the void.

As C++ is extended to support data parallelism (e.g., through the introduction of data-parallel for loops), there will be new back ends for C++ compilers that can target GPUs (and eventually FPGAs), and some of the programs written today in CUDA will be written tomorrow in data-parallel versions of C++ and other mainstream languages such as Java and C#. To extract the best performance from GPUs, however, one will still have to use systems such as CUDA that expose low-level architectural features; but for many classes of algorithms and users, the higher-level compilation path from data-parallel conventional languages will be sufficient. Ultimately, we will need to devise new languages that provide effective memory models for modern computing systems.²

LANGUAGE-NEUTRAL MULTITARGET DATA-PARALLEL PROGRAMMING

To meet today's need for a higher-level data-parallel programming system that can target heterogeneous systems, one can look to the Accelerator system from Microsoft.¹³ It allows certain kinds of data-parallel descriptions to be written once and then executed on three different targets: GPUs, multicore processors using SSE3 vector instructions, and FPGA circuits, as illustrated in figure 1. (Microsoft Accelerator can be downloaded from <http://research.microsoft.com/en-us/projects/Accelerator/>.)

In general we cannot hope to devise one language or system for programming heterogeneous systems that allows us to compile a single source into efficient implementations on wildly different

FIGURE 1 **Compilation from a Single Description to Multiple Targets**

computing elements such as CPUs, GPUs, and FPGAs. Such parallel-performance portability is difficult to achieve. If the problem domain is sufficiently constrained, however, it is possible to achieve good parallel performance from a single source description. Accelerator achieves this by constraining the data types used for parallel programming (to whole arrays that cannot be explicitly indexed) and by providing a restricted set of parallel array access operations (e.g., in order, in reverse, with a stride, shifted, transposed).

Not all kinds of data-parallel algorithms are a suitable match (e.g. matrix inversion does not fit this model). For many classes of algorithms (e.g., stencil-style computations⁸), however, the Accelerator system makes it possible to compile a single description to three very different kinds of hardware and obtain good performance. One can always make a faster implementation using CUDA, SSE3 intrinsics, or Verilog, but this performance improvement comes at significantly higher design cost.

A key aspect of Accelerator is that it is not a new language. The system works just as a library that can be used from any language that supports linking with a C calling interface under Windows. Applications that use the Accelerator system have been written in several languages including C++, C#, F#, and the functional language Haskell.

Accelerator is an example of an EDSL (embedded domain-specific language). The Accelerator system provides the user with an abstract programming language that encodes notions of arrays and whole array operations and certain memory transforms. This abstract language is embedded in a conventional concrete language, and programs of the abstract language are expressed as data structures in the concrete language. This trick is a valuable technique for programming heterogeneous systems, allowing you to write data-parallel descriptions in a language-agnostic way and cross-compile to a variety of hardware architectures.

Consider the task of performing an element-wise data-parallel addition of two arrays: [1, 2, 3, 4, 5] and [6, 7, 8, 9, 10]. This very simple Accelerator program is shown here as a C++ program:

```
#include <iostream>
#include "Accelerator.h"
#include "DX9Target.h"

using namespace std;
using namespace ParallelArrays;
using namespace MicrosoftTargets;

int main()
{
    // Create a GPGPU computing resource
    DX9Target *tgtDX9 = CreateDX9Target();

    // Declare some sample input arrays
    float xvalues[5] = {1, 2, 3, 4, 5};
    float yvalues[5] = {6, 7, 8, 9, 10};

    // Create data-parallel versions of inputs
```

```

FPA x = FPA(xvalues, 5) ;
FPA y = FPA(yvalues, 5) ;

// Specify data-parallel computation
FPA z = x + y ; // Computation does not occur here...

// Allocate space for the result array
float* zvalues = (float*) malloc (5 * sizeof(float)) ;

// Execute the data-parallel computation on the GPU
tgtDX9->ToArray(z, zvalues, 5); // z = x + y is now evaluated

// Write out the result
for (int i = 0; i < 5; i++)
    cout << zvalues[i] << " " ;
cout << endl ;

return 0 ;
}

```

When executed this program produces the output:

```
7 9 11 13 15
```

This regular C++ program can be compiled with the unmodified Visual Studio compiler and linked with the Accelerator library. When it executes it generates GPU code (using the DX9 system) at runtime on the host processor by JITing (using just-in-time compilation), transfers the generated code and data to the GPU for execution, and then transfers the results back to the host for display. The JITing model allows the system to hide many of the low-level details about the generation of GPU code and the communication with the GPU hardware. It is important to note that the data-parallel addition of the arrays *x* and *y* is specified by using an overloaded definition for the “+” operator that builds a data structure in the heap to express the intended computation:

FPA operator+ (FPA a1, FPA a2)

This code represents the input and output data-parallel arrays in “texture memories” and uses a data-parallel add instruction over these arrays. The user of the Accelerator system is entirely insulated from the graphics-level view of the computation.

To run the same computation as SSE3 vector instructions on a multicore processor split across multiple processing cores, you would write exactly the same program but specify a different target. To emphasize the language-neutral aspect of Accelerator, this example switches the source language to C# for the multicore SSE3 version:

```

using System;
using Microsoft.ParallelArrays;
namespace AddArraysPointwiseMulticore
{
    class AddArraysPointwiseMulticore
    {
        static void Main(string[] args)
        {
            var x = new FloatParallelArray (new[] {1.0F, 2, 3, 4, 5});
            var y = new FloatParallelArray (new[] {6.0F, 7, 8, 9, 10});
            var multicoreTarget = new MulticoreTarget();
            var z = x + y;
            foreach (var i in multicoreTarget.ToArray1D (z))
                Console.Write(i + " ");
            Console.WriteLine();
        }
    }
}

```

When this program is run, it dynamically generates (JITs) SSE3 vector instructions and splits the data across multiple cores, then fires up vector instructions on each core to deliver exactly the same result as the GPU version.

MAPPING EXPLICIT DATA-PARALLEL COMPUTATIONS ONTO FPGAS

When the performance requirements cannot be met by a multicore SSE3 implementation or by a GPU implementation, you can try to produce an FPGA implementation by writing the same description of the parallel computation and specifying the use of an FPGA target. Again, to emphasize the language-neutral aspect of Accelerator this example switches the source language to F#:

```

open System
open Microsoft.ParallelArrays
let main(args) =
    let x = new FloatParallelArray (Array.map float32 [|1; 2; 3; 4; 5 |])
    let y = new FloatParallelArray (Array.map float32 [|6; 7; 8; 9; 10 |])
    let z = x + y
    use fpgaTarget = new FPGATarget("adder")
    fpgaTarget.ToArray1D(z)

```

Note that this example does not write out the results of executing the data-parallel description on an FPGA chip. We cannot currently support a JITing model for the FPGA Accelerator target because the design tools that process hardware descriptions into FPGA programming bitstreams (the machine code of FPGA circuits) typically take a very long time to execute (tens of minutes or even hours).

This target works in an offline mode and generates a VHDL (VHSIC hardware description language) hardware description that is deposited in a file (`adder.vhd`) that can be used as input to special design tools provided by FPGA vendors.

A more realistic example of an Accelerator program is a convolver that computes a weighted average over a stream or image, as shown in figure 2. A straightforward way to implement this algorithm is to represent the computation with nested for loops which use explicit array indexing to capture the notion of a sliding window. However, this is a poor starting point for a system that wants to generate a parallel implementation or one that can be targeted to a heterogeneous collection of processing cores or digital hardware because it is difficult to map the arbitrary array indexing operations into efficient memory access operations on various targets.

FIGURE 2 One-Dimensional Convolution

$$y_t = \sum_{k=0}^{N-1} a_k x_{t-k}$$

FIGURE 3 The Shift Memory-Access Transform

A much better way to express this computation is in terms of a whole array operation combined with explicit memory-access transformations. An example of a memory transform operation is *shift*, which is shown in figure 3. This operation takes an input array *x* and produces an output array that may be shifted right (-1) or left (+1).

Figure 4 shows an application of a shift operation used to describe successively shifted versions of the input array, after which whole array multiplication is performed in parallel followed by a parallel reduction (of add operations) to yield the convolved output.

In Accelerator the whole array description of the convolution operation can be expressed as shown here:

```
using Microsoft.ParallelArrays;
using A = Microsoft.ParallelArrays.ParallelArrays;
namespace AcceleratorSamples
{
    public class Convolver
    {
        public static float[] Convolver1D(Target computeTarget,
            float[] a, FloatParallelArray x)
        {
            var n = x.Length;
            var y = new FloatParallelArray(0.0f, new [] { n });
            for (int i = 0; i < a.Length; i++)
                y += a[i] * A.Shift(x, -i);
            float[] result = computeTarget.ToArray1D(y);
            return result;
        }
    }
}
```

FIGURE 4

One-Dimensional Convolution Expressed with Shifts

shift(x,0)	7	2	5	9	3	8	6	4	→	a[0] * shift(x,0)	14	4	10	18	6	16	12	8	
shift(x,-1)	7	7	2	5	9	3	8	7	→	a[1] * shift(x,-1)	35	35	10	25	45	15	40	30	
shift(x,-2)	7	7	7	2	5	9	3	8	→	a[2] * shift(x,-2)	49	49	49	14	35	63	21	56	
shift(x,-3)	7	7	7	7	2	5	9	3	→	a[3] * shift(x,-3)	28	28	28	28	8	20	36	12	
shift(x,-4)	7	7	7	7	7	2	5	9	→	a[4] * shift(x,-4)	21	21	21	21	21	6	15	27	
											↓	↓	↓	↓	↓	↓	↓	↓	
											+	+	+	+	+	+	+	+	
											y =	147	137	118	106	115	120	124	133

This description is parameterized on the “target” (i.e., it can be used to generate multithreaded SSE3 instructions, GPU code, or FPGA circuits, depending on the actual value of `computeTarget`). Note that the array of weights `a` is a regular C# array, because this represents compile-time constants rather than data that is only known at runtime. The other array parameter `x` has a different type: `FloatParallelArray`. This represents a data-parallel array that may live in a different address space from the host C# program (e.g., on a memory bank of the GPU or on an embedded memory on an FPGA chip). These arrays do not permit explicit array indexing but instead offer memory-transform operations such as `Shift`.

To illustrate the kind of performance improvement that might be expected from such an approach, the Accelerator version of a two dimensional convolver computation is compared against a sequential version written as two separable passes.

The Accelerator version works by applying two one-dimensional convolutions and takes as a parameter the specific target to be used for the data-parallel execution of this algorithm:

```
public static float[,] Convolver2D(Target computeTarget, float[] a, float[,] x)
{
    var xpar = new FloatParallelArray(x);
    var n = x.GetLength(0);
    var m = x.GetLength(1);
    var zpar = new FloatParallelArray(0.0f, new[] { n, m });
    var shiftBy = new[] { 0, 0 };
    for (var i = 0; i < a.Length; i++)
    {
        shiftBy[1] = -i;
        zpar += a[i] * A.Shift(xpar, shiftBy);
    }
    shiftBy[1] = 0;
    var ypar = new FloatParallelArray(0.0f, new[] { n, m });
    for (var i = 0; i < a.Length; i++)
    {
        shiftBy[0] = -i;
        ypar += a[i] * A.Shift(zpar, shiftBy);
    }
    return computeTarget.ToArray2D(ypar);
}
```

The sequential version was run on a computer with two Intel Xeon E7450 processors, each clocked at 2.4 GHz. The machine had 32 GB of memory and ran Windows Server 2008 R2 Enterprise. The Accelerator version was run using the SSE3 multicore target on the same machine, testing the same code using two different graphics cards: an ATI Radeon HD5870 and an Nvidia 470 GTX using the DX9 target. The input matrix for the convolution was set to 8,000 by 8,000, and the experiment varied the size of the convolution kernel. The speedups achieved over the sequential baseline version are shown in Figure 5. The execution time for each of the Accelerator experiments includes the end-

FIGURE 5

Performance Improvement from Single Source Mapped to Multiple Targets

to-end time that measures the cost of JITing and data movement to and from the GPU or to and from thread-local state in the SSE3 case.

Figure 5 also shows that as the convolution kernel gets larger both the SSE3 multicore target and the GPU targets show improving performance. Cache effects, however, cause the speedup of the SSE3 multicore target to degrade after the kernel size grows to be larger than 40. This experiment shows the viability of using a single-source description that can be mapped to two very different targets to yield meaningful performance improvements.

THE IMPORTANCE OF SPECIFYING COMMUNICATION

A key aspect of designing descriptions is the ability to describe memory-transform operations at a suitable level of abstraction. Descriptions that make too many assumptions about memory organization and access (e.g., random array indexing) are difficult or impossible to port to other architectures that have wildly different memory-access economics.

Shift is a powerful memory-access abstraction because it reveals what information to reuse so it is worth caching, storing in a register, or placing into local shared memory on a GPU (figure 6). For example, when the FPGA target is used to implement the convolver, the shift operations are used to infer delays (and anti-delays) that allow you to reach each data item once but use it three times. (In an actual implementation extra delays are added to cancel the anti-delays, which are hard to come by in the real world.) For a two-dimensional convolver this advantage is even more pronounced.

FIGURE 6

Shift: Reusing Rather than Rereading Data

The delays introduced by the shift operator can be redistributed through the circuit to yield a transposed implementation of the convolver automatically (shown in figure 7). This is a particularly good fit for FPGA technology because of the abundance of registers for implementing pipeline delays. A typical FPGA implementation of this circuit produces a system that has a delay of around five nanoseconds (for integer values), which allows 200 million elements to be convolved in a second. Many of these convolvers can be laid down on an FPGA circuit, and each can run in parallel.

MULTIPLE TARGETS

The Accelerator model is limited to mapping a single description completely onto one of several targets. A natural extension is to imagine a system that compiles a single data-parallel description onto a mixture of processing elements. For example, certain subexpressions could be mapped to GPU hardware and others could be mapped to FPGA hardware, and the compilation process would also generate the software needed to orchestrate the movement of data between the different types of hardware elements and coordinate their execution. This capability is useful for producing a family

FIGURE 7

Automatically Generated Transposed 1D Convolver

of implementations for given functions that represent different price/performance points or different energy/latency tradeoffs.

Mapping a single computation onto a heterogeneous set of cooperating processing elements can be taken one step further by allowing a computation to migrate from one type of processing hardware (say, a GPU) to another (say, an FPGA) in a transparent manner.

WHAT WE MIGHT SEE IN 18 MONTHS

Turning programs into circuits has not been a glorious form of alchemy and has failed to produce what many developers want (a technique for genuinely converting real software into circuits). Instead it has merely provided digital designers with the ability to write highly stylized circuit descriptions that are a heartbeat away from low-level register transfer-level descriptions, which are usually cast in VHDL or Verilog. These techniques may be useful for improving the programming productivity of hardware engineers, but they do not form the basis of an approach that effectively converts realistic software into hardware circuits (or code for other kinds of heterogeneous computing elements).

Just as some alchemists eventually uncovered enough scientific and theoretical basis for their craft and then rebranded themselves as socially acceptable chemists, there are signs that a similar transmutation is occurring in the field of genuine high-level synthesis. Two very encouraging projects that seriously tackle the task of converting real programs into circuits are the Liquid Metal project at IBM, based on the Lime language that extends Java with special concurrency constructs,³ and the Kiwi project at the University of Cambridge by Dr. David Greaves and at Microsoft Research Cambridge (which works on unmodified standard .NET programming languages and relies on multi-threaded programs to describe important parallel execution economics).⁶

Multicore processors and GPU architectures are beginning to move closer to each other. Intel's Sandy Bridge processors have AVX (Advanced Vector Extensions), which provide data-parallel operations over 256-bit registers. AMD's Fusion program promises ever tighter integration between CPUs and GPU cores on the same die. There is already a proof-of-concept single-package chip that combines an Intel Atom processor and an Altera FPGA. This could be the stepping-stone to systems that put FPGA-like logic on the same die as processor cores. The shift to a heterogeneous world looks inevitable.

Although the focus of this article has been heterogeneous *processing*, it is important to note that currently evolving heterogeneous systems consist of many other kinds of heterogeneous elements (e.g., storage, memory, and networking). The management of different kinds of storage (e.g., physical versus solid state), memory (e.g., dealing with different types of memory with different access economics and latencies and reliability characteristics), and networking (e.g., optical interconnects versus copper and wildly varying protocols) imposes formidable challenges to the design of systems software and application software for modern heterogeneous systems.

FURTHER OUT...

There has also been progress in converting C programs that manipulate heap data structures with pointers into circuits using recent advances in shape analysis and region types. The conventional wisdom in the high-level synthesis field would lead one to believe that such transformations are impossible, but today you can deduce a surprising amount of information symbolically about what is happening in the heap of C programs and can determine with a far greater degree of fidelity which

parts of the heap any given instruction may touch (compared with conventional point-to analysis).

These advances in the theory of program analysis arm us with the power tools needed to make the transmutation of programs into circuits (or GPU code) possible. These techniques allow a blurring of the distinction between hardware and software because they permit a computation cast as software working over a heap using pointers to be recast as a computation over an array (when certain bounds can be automatically computed) and then the code to be remapped to GPUs or FPGAs. This ability to convert an algorithm to preserve its meaning but refactor it to work over a different representation of its data that is more appropriate for a different execution environment is a key weapon that makes the retargetable programming of heterogeneous systems possible.

Researchers such as Luca Cardelli⁵ and Masami Hagiya⁷ are devising techniques for programming with DNA, pushing the frontiers of what it is possible to compute with. Such research may shed light on how to build systems that can not only repair themselves but also assemble themselves. Although mainstream programmers are not likely to be programming with DNA anytime soon, such projects remind us that there are many things in the world that have the capability to compute. We face an exciting challenge in working out how to program them.

CONCLUSION

Programming until now has largely focused on cajoling CPUs into implementing our algorithms. This has involved significant assumptions about the economics of the underlying execution architecture and in particular about the architecture of the memory system. Increasingly, we will need to program computational engines which have radically different architectures from regular CPUs in order to meet our requirements for performance improvement or to reduce energy consumption.

Mapping the same computation onto several different computing elements typically involves re-writing algorithms to achieve an efficient execution on each architecture. This typically involves using a different programming language and a very different model of execution compared to the original algorithm that was developed for a CPU.

This is an expensive operation and although it will be necessary in many cases it is possible to identify a constrained class of problems that share some key architectural assumptions. This knowledge can then be exploited to permit an automatic compilation from one description onto multiple targets e.g. CPU vector instructions, GPUs, and FPGAs.

The Accelerator system is an example of this approach that provides an embedded domain-specific language for stencil-style computations that can be written in C, C++, or any language that supports a C calling interface, e.g. C#, F#, or Haskell. In the case of Accelerator, the constraints are the use of data-parallel arrays that permit only whole array operations (no arbitrary array indexing) and the provision of explicit memory transform operations (e.g. shifting, rotation, strides). This avoids baking in assumptions about a particular memory model and permits the efficient mapping of Accelerator data-parallel computations onto very different execution targets.

As the number of heterogeneous processing elements grows we will need to develop a collection of such embedded domain specific languages each of which captures the essence of the computational requirements for a specific domain (e.g. stencil computations, XML parsing, regular expression matching). For each embedded domain-specific language we can then implement a mechanism for compiling descriptions onto a specific execution architecture.

By combining embedded domain specific languages which are used through library calls in

mainstream programming languages we can take an important step towards developing mechanisms that make the programming of heterogeneous systems tractable while avoiding the cost of re-implementation or the cost of developing and adopting new languages.

REFERENCES

1. Abadi, M., et al. 2011. Semantics of transactional memory and automatic mutual exclusion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33(1).
2. Adve, S. V., Boehm, H-J. 2010. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM* 53(8).
3. Auerbach, J., et al. 2010. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 10).
4. Borkar, S., Chien, A. A. 2011. The future of microprocessors. *Communications of the ACM* 54(5).
5. Cardelli, L. 2009. Strand algebras for DNA computing. *Natural Computing* 10(1): 407.
6. Greaves, D., Singh, S. 2010. Designing application-specific circuits with concurrent C# programs. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign.
7. Hagiya, M. 1998. Towards autonomous molecular computers. Proceedings of the 3rd Annual Genetic Programming Conference.
8. Lesniak, M. 2010. PASTHA - parallelizing stencil calculations in Haskell. Proceedings of the Fifth ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming.
9. Madhavapeddy, A., Singh, S. 2011. Reconfigurable data processing for clouds. MSDN Blogs; http://blogs.msdn.com/b/satnam_singh/archive/2011/01/18/reconfigurable-data-processing-for-clouds.aspx.
10. Nvidia Corporation. 2007. Nvidia CUDA (compute unified device architecture); <http://developer.nvidia.com/cuda>.
11. Owens, J. D., et al. 2008. GPU computing. *Proceedings of the IEEE* 96(5).
12. Shavit, N. 2011. Data structures in the multicore age. *Communications of the ACM* 54(3).
13. Tarditi, D., Puri, S., Oglesby, J. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. Architectural Support for Programming Languages and Operating Systems.
14. Trancoso, P., Othonos, D., Artemiou, A. 2009. Data-parallel acceleration of decision support queries using Cell/BE and GPUs. Proceedings of the 2009 ACM International Conference on Computing Frontiers.
15. Xu, L., Wan, J. W. 2008. Real-time intensity-based rigid 2D-3D medical image registration using rapidmind multi-core development platform. Proceedings of the 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

SATNAM SINGH (s.singh@acm.org) is a researcher at the Microsoft Research Cambridge UK laboratory, where he investigates high-level design techniques for low-level systems. He also holds the Chair of Reconfigurable Computing at the University of Birmingham. (<http://research.microsoft.com/~satnam>; <http://www.cs.bham.ac.uk/~singhsu/>)

© 2011 ACM 1542-7730/11/0600 \$10.00