Programming Future Parallel Architectures with Haskell and Intel ArBB

Bo Joel Svensson
Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: joels@chalmers.se

Ryan Newton
Intel Corporation
Hudson, MA
Email: rrnewton@gmail.com

***Abstract*—New parallel architectures, such as Cell, Intel MIC, GPUs, and tiled architectures, enable high performance but are often hard to program. What is needed is a bridge between high-level programming models where programmers are most productive and modern parallel architectures. We propose that that bridge is Embedded Domain Specific Languages (EDSLs).**

One attractive target for EDSLs is Intel ArBB, a virtual machine for parallel, vectorized computations. We propose to wed ArBB with the functional programming language Haskell, using an EDSL that generates code for the ArBB VM. This Haskell integration provides added safety guarantees compared to other ArBB interfaces. Further, our prototype, Harbb, is one of the first EDSL implementations with optimized backends for multiple parallel architectures (CPU, NVIDIA GPU, and others), allowing portability of source code over devices and their accelerators.

I. Introduction

Are radical new parallel architectures market-feasible if they require significant changes for programmers? The jury is out. In recent years we have seen difficult-to-program chips suffer (e.g. Cell) and GPU vendors strive to enable more traditional programming features [1] (e.g. C++). There is an increasing tension between ease of programming and efficiency.

The tension shows across diverse chip markets. For example, small embedded devices are most power efficient if their processors and operating systems omit programming features such as virtual memory and threads [2]. At the other end of the power spectrum, GPU's graphics performance may suffer due to inclusion of hardware to ease GPGPU programming. In short, there is an opportunity cost to including extra hardware for programmability.

In this paper, we argue that a specific technique holds the greatest promise of solving the programmability dilemma. Domain-specific languages, *embedded* within general purpose languages (EDSLs) enable familiar programming models *and* flexible mapping onto new hardware. The key to having this cake and eating it too is *metaprogramming*. Familiar programming features are present, but are eliminated at an intermediate (metaprogram evaluation) phase and therefore do not reach the parallel hardware itself.

In pursuit of this vision, we offer a new EDSL implementation, called Harbb, that combines existing systems, Accelerate [3] and ArBB [4], to produce a unified high-level programming environment equally suited to multicore, vectorized CPUs, as to GPUs and other accelerators (such as Intel MIC chips [5]). Harbb is a single EDSL implementation with independently optimized backends by different teams; namely, the ArBB backend for CPU/MIC, and a CUDA backend for NVIDIA GPU. This makes Harbb an appealing platform for fair CPU/GPU comparisons, as well as a compelling programming model for single-source portable performance across a range of parallel architectures, present and future.

II. Embedded Domain-Specific Languages

Domain-specific languages—from Makefiles and LaTeX to Matlab—are almost too ubiquitous to notice. Most relevant to our purposes, domain-specific languages (DSLs) that target a narrow domain and expose communication patterns to the compiler have achieved performance-portability across a wide range of parallel architectures. StreamIt[6] is a good example.

DSLs may start out simple and focused, but if they gain popularity they quickly grow in complexity to rival full-blown languages. Feature creep can make DSL implementations complex and expensive to maintain. Further, non-standard DSL syntax and features present a learning curve for users. In the last ten years an attractive solution to this dilemma has emerged: *embed* each DSL into a general-purpose host language that can provide common functionality with familiar syntax and semantics.

When embedding, host language programs generate DSL programs; the *deeper* the embedding, the more integrated the DSL into the syntax and type-system of the host language. A key host-language feature for embedding is that language constructs can be overloaded to operate over *abstract syntax trees* (ASTs)[1]. For example, the following simple function operates on scalars:

```
float f(float x) { return (2*x - 1); }
```

But simply by changing the types, `f` might be lifted to operate on *expressions* (which, when evaluated, will yield `floats`):

```
exp<float> f(exp<float> x) {
  return (2*x - 1);
}
```

[1]This ad-hoc polymorphism is accomplished, for example, through operator-overloading in C++ or type classes in Haskell.

A common arrangement is for the host language program to execute at runtime but to generate ASTs that are executed by a just-in-time (DSL) compiler. This use of metaprogramming (program generation) differs from the more common usage of preprocessors and macros, which typically add extra phases of computation *before* compile time—increasing the number of compile-time rather than runtime phases.

One reason that EDSLs are good for productivity is that the programmer gains the software engineering benefits of the host language (object-orientation, higher-order-functions, modules, etc), while not paying the cost at runtime for additional layers of abstraction or indirection. Indeed, the embedded languages for performance-oriented EDSLs are often simple, first-order languages without pointers [7], [3].

As a research area, EDSLs and two-stage DSLs have been actively pursued for at least a decade [8], [7] but are gaining steam recently [9], [10] and are beginning to appear in commercial products [4]. Further, EDSL techniques have spread beyond their origin in the programming languages community. For example, both Stanford's Parallel Programming Laboratory (PPL) and the Berkeley Parlab are creating EDSLs as their flagship parallel programming solutions for domains such as machine learning and rendering [10]. Moreover, EDSLs need not be hosted by esoteric research languages—Intel's ArBB embeds an array language in C++ and Berkeley's Copperhead [9] generates CUDA from simple Python code.

For the remainder of this paper we will focus our discussion on the Intel ArBB VM, a virtual machine for just-in-time generation of vector codes, which implements a restricted domain-specific language of array computations. In this paper we introduce *High-level ArBB*, (Harbb), an EDSL that internally uses the ArBB VM. While Intel's ArBB package already includes an EDSL targeting the VM (for C++), Harbb offers additional advantages, including more succinct programs and additionally safety guarantees—namely, complete deterministic-by-construction parallel programs (across both host and VM languages).

III. HARBB = ARBB + ACCELERATE

Our first Harbb prototype adapts an existing EDSL called *Accelerate* (Data.Array.Accelerate). Accelerate targets high-level data-parallel programming in Haskell. Previous work on Accelerate has focused on developing a CUDA-backend for GPU programming. In this paper we describe our effort to retarget Accelerate to ArBB.

With respect to determinism guarantees, the existing Intel ArBB product represents an integration of the safe (ArBB VM) with the unsafe (C++). In the Haskell context, because purely functional computations are guaranteed deterministic (even when executed in parallel), and because ArBB computations invoked by Haskell functions are themselves free of side-effects, Harbb achieves a guarantee of determinism for complete programs that combine both Haskell computation and ArBB VM computation.

The Accelerate programming model consists of collective operations that can be performed over arrays, together with a simple language of scalar expressions—in the current release, the Haskell type system enforces that parallelism not be nested. Accelerate's collective operations include *Map* (akin to parallel for loops) *ZipWith* (a generalization of elementwise vector addition) and *Fold* (sum generalized)—familiar operations for programmers versed in the functional paradigm. All of these collective operations are easily parallelizable.

```
dotProd (xs :: Vector Float)
        (ys :: Vector Float) =
  let xs_ = use xs
      ys_ = use ys
  in  fold (+) 0 (zipWith (*) xs_ ys_)
```

The Accelerate code listing above specifies a function that takes two 1-dimensional arrays as inputs (of type `Vector Float`, e.g. a vector of floats). The result of the function is a single scalar. The `use` function is applied to an array to convert it for use in the collective operations provided by Accelerate; `use` may result in copying the array to, for example, the GPU in the case of Accelerate's CUDA backend.

After applying `use` to bring in input data, the programmer then constructs a data-parallel program from collective operations. Above, `fold` is a function that takes three arguments, here those arguments are `(+)`, 0 and `zipWith (*) xs_ ys_`. `zipWith`, in turn, is a function taking three arguments, `(*)`, `xs_` and `ys_`. The `zipWith` operation here applies pairwise multiplication to the two arrays and the `fold` sums up all the elements into a single scalar.

Accelerate's CUDA backend implements collective operations using a hand-tuned "skeleton" for each operation (and possibly for different hardware versions). The kernel—for example, the function to mapped over the dataset—is instantiated into the body of the skeleton code. The resulting program is compiled using the NVIDIA CUDA compiler and dynamically linked into the running Haskell program.

IV. INTEL ARRAY BUILDING BLOCKS (ARBB)

The operations exposed by ArBB are similar to those of Accelerate. In ArBB, parallel computations are expressed using a set of built-in primitives. All vectorization and threading is managed internally by ArBB. The programmer uses collective operations with a clear semantics such as `add_reduce` that computes the sum of the elements in a given array. ArBB also has language constructions for control flow, conditionals and loops. These operations have their usual sequential semantics and are not parallelized by the system, rather, only specific collective operations are executed in parallel.

Today's existing ArBB product is embedded in C++ and provides special types for scalars and arrays (e.g. `dense<f32>` rather than `vector<float>`). Using ArBB/C++ to express the dot product computation can be done as follows:

```
void dot_product(const dense<f32>& a,
                 const dense<f32>& b,
                 f32& c)
{
  c = add_reduce(a * b);
}
```

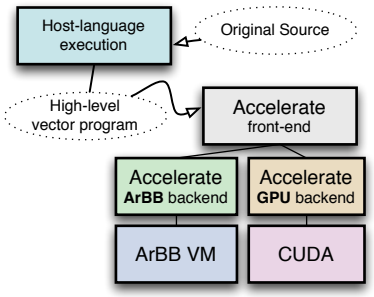

Fig. 1. Architecture of the Harbb system.

Note that arithmetic operators such as (*) are overloaded for to operate on arrays as well as scalars (e.g. `a * b` above). Thus, the above C++ function multiplies two arrays before summing them with `add_reduce`:

The code listing below is indicative to the amount of glue code needed to invoke an ArBB computation[2]. It shows how the dot product code is launched using `call` and how data is bound, `bind`, for use in ArBB.

```
int main()
{
  double a[SIZE];
  double b[SIZE];
  for ( int i = 0; i < SIZE; ++i) {
    a[i] = ...; b[i] = ...;
  }
  dense<f32> va, vb;
  f32 vc;
  bind(va, a, SIZE);
  bind(vb, b, SIZE);
  call(dot_product)(va, vb, &vc);
  ...
}
```

The model provided by Accelerate is slightly richer than that of ArBB. Even the two very simple `dot_product` examples above manage to illustrate this. In Accelerate there is a more general reduction primitive called `fold` where in ArBB there are specific reductions, `add_reduce`, `mul_reduce` and so on. Not visible in these small examples is another difference, Accelerate operations are generalized to arbitrary dimensions while ArBB operations are limited to 1, 2, and 3 dimensions (and 0, i.e. scalar). These differences aside, the ArBB and Accelerate programming models are very similar.

V. Implementation of Harbb

Using Accelerate and ArBB together, we propose a layered architecture for Harbb, pictured in Figure 1. The host language execution (via Haskell in this case) executes the programmer's source code, generating a vector program in the restricted language of the Accelerate front-end. Then either Accelerate backend—ArBB or CUDA—may be used. The bottom layer of the ArBB backend consists of a direct mapping of the ArBB virtual machine API (VMAPI) into Haskell including one-to-one bindings for each C function. To [partially] automate the creation of these bindings we used the C2HS system [11]. The ArBB/Haskell bindings are very low-level. The idea is that the ArBB/Haskell bindings should be used to implement backends for higher-level data-parallel EDSLs.

To address the functionality mismatch between Accelerate and ArBB, when possible we reencode the users Accelerate program using existing ArBB mechanisms. Our prototype does not support 100% of Accelerate's compute model (for example, only supporting up to three-dimensional arrays), but the remaining functionality can be mapped onto ArBB in time using known methods—for example, our compiler could map higher dimensional arrays onto a specific lower-dimensional data-layout.

One example of a functionality discrepancy bridged by our implementation is reductions. As mentioned in IV, Accelerate allows the programmer to reduce using an arbitrary associative function, but ArBB has only built-in reductions with fixed operations (add, multiply, xor, etc). We plan to provide general reductions in the ArBB backend by a two-fold strategy:

1) Attempt to map an Accelerate reduction directly onto an ArBB primitive such as `add_reduce`.
2) Apply a general reduction technique based on $log(N)$ map operations over successively halving array sizes. Essentially, cut the array in half, combine the halves, repeat. In [12] this approach is explained in the context of CUDA.

In our current experiments, approach (2) is significantly slower. Therefore, the ideal would be that ArBB exposed general reduction directly in its programming model. We expect this functionality to be added in future releases. In the meantime we plan to explore a technique that would allow us to maximize the number of situations in which (1) above applies. Namely, a reduce operation can often be *factored* into a map followed by a reduce. For example, a reduction that multiplies each input number by a coefficient and sums the results can be split into a map phase for the multiplication followed by the built-in `add_reduce` operator.

Another choice faced by our implementation is the granularity at which the ArBB JIT is invoked. Specifically, should each collective operation result in its own call to the ArBB JIT (in ArBB terminology, *immediate-mode*, akin to the pre-OpenGL 3.0 immediate mode), or should multiple collective operations be placed together inside an ArBB function and passed to the JIT? We will call the latter approach *retained-mode*.

Retained-mode generally offers performance benefits; a bigger chunk is given to the JIT compiler, enabling cross-optimization between collective operations. Our prototype Accelerate backend uses ArBB in a combination of immediate- and retained-mode. The main collective operations are compiled using the retained-mode. For example, in the case of a `map f` operation first the function to be mapped is created and compiled using retained-mode then a small *mapper* function is also created and compiled using retained-mode. Between the collective operations the backend needs to perform data management and copying, which are performed in immediate-mode. It is our belief that Harbb would benefit from using retained-mode exclusively but we leave that as future work.

[2]In OpenCL and CUDA the glue code situation is even worse

```
blackscholes (xs :: Vector (Float,Float,Float)) =
  map kernel (use xs)

kernel x =
  let (price, strike, years) = unlift x
      r     = 0.02 – riskfree constant
      v     = 0.30 – volatility constant
      sqrtT = sqrt years
      d1    = (log (price / strike) +
                    (r + 0.5 * v * v) * years) /
              (v * sqrtT)
      d2    = d1 - v * sqrtT
      cnd d = d >* 0 ? (1.0 - cndfn d, cndfn d)
      cndD1 = cnd d1
      cndD2 = cnd d2
      expRT = exp (-r * years)
  in lift ( price * cndD1 -
            strike * expRT * cndD2
          , strike * expRT * (1.0 - cndD2) -
            price * (1.0 - cndD1))

cndfn d =
  let poly  = horner coeff
      coeff = [0, 0.31, -0.35, 1.78, -1.82, 1.33]
      rsqrt = 0.39894228040143267793994
      k     = 1.0 / (1.0 + 0.2316419 * abs d)
  in rsqrt * exp (-0.5*d*d) * poly k

horner coeff x =
  let madd a b = b*x + a
  in  foldr1 madd coeff
```

Fig. 2. Complete code listing for a Black-Scholes function expressed in Haskell syntax using the Accelerate and Harbb libraries. Invocations of the functions `use`, `lift` and `unlift` represent additional boilerplate added for conversion in and out of Accelerate types. Specifically, `lift` and `unlift` convert tuples and handle the fact that Accelerate arrays of tuples are really implemented as tuples of arrays. Otherwise, the program is identical to a plain Haskell implementation.

VI. Preliminary Results

Black-Scholes option pricing is a finance-related benchmark that has been used in similar DSLs targeting GPUs [3], [13]. Since we are re-using the Accelerate front-end, we can directly use the Black-Scholes benchmark that is shipped with that system. Figure 2 shows the complete code listing for an Accelerate Black-Scholes function which can be executed on GPUs or any processor targeted by ArBB.

The kernel of this algorithm performs arithmetic on triples of floating point numbers, creating pairs of floats as results. The problem is embarrassingly parallel, consisting of independent computations for every element of an array (a map).

Figures 3 and 4 show preliminary results obtained on the Black-Scholes benchmark. The figures where obtained on a system with a 4-core Intel Core I7 975 machine with HyperThreading. The GPU used was a NVIDIA GTX480.

Figure 3 shows running times obtained when JIT-time is included. JIT compilation time is much larger in the CUDA backend than the ArBB one. Part of this difference can be attributed to the fact that the CUDA backend calls an external compiler (`nvcc`), which takes its input in a file and runs in a separate process. ArBB, on the other hand, has a library interface to its JIT-compiler. Figure 4 shows results obtained when pre-compiling the CUDA functions eliminating

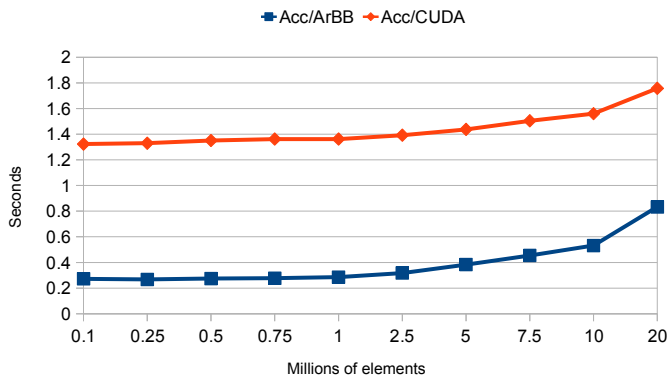

Fig. 3. Running time experiments including JIT-time.

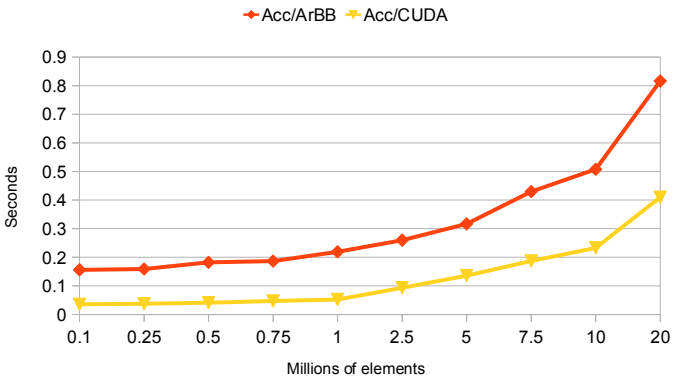

Fig. 4. Running time experiments JIT-time excluded.

JIT overhead. In Accelerate, this happens automatically when the same kernel is invoked repeatedly, because the Accelerate CUDA backed uses a caching scheme to avoid unnecessary JIT invocations. (The caching functionality has not yet been duplicated in the ArBB backend.)

These early results demonstrates the principle that even when a kernel executes with higher throughput on a GPU, in a particular program it is difficult to decide whether a computation is worth moving to a GPU, incurring extra data-movement and possibly extra JIT compilation [14]. Specifically, we see that the Accelerate Black-Scholes program from Figure 2 performs better on the CPU if it executes once (even on a large window of data) whereas the GPU would yield better performance in a sustained series of executions. Because both CPU and GPU execution may be desirable—and the selection may be dynamic—it is beneficial to have a single source code that is portable across both.

VII. Related Work

OpenCL [15] is a programming model very similar to CUDA but with the aspiration to offer both acceleration of computations on GPUs or to multicore CPUs. OpenCL JIT compiles the kernels for the particular hardware available and is in that sense similar to ArBB. OpenCL programs are relatively low-level and require a large amount of boilerplate to create and invoke. In this sense they occupy a very different niche than Accelerate.

Microsoft Accelerator [16] is an embedded language with similar aspirations as ArBB, that is, to target a diverse range of architectures using the same source code. Accelerate can be used from the C# language or the functional F# language and targets GPUs or of CPUs and their vector units.

Many CPU and GPU comparisons, and some CPU/GPU workload partitioners [17], rely on redundant hand-written versions of all kernels (though some systems like Qilin [18] allow a single source code). It is difficult in this kind of scenario to make fair comparisons, controlling for the amount of effort put into the respective implementations. For example, comparing unoptimized serial CPU implementations vs. GPU ones is not informative [19]. In Harbb controlling for effort need happen only once—both CUDA and ArBB are independently optimized by their respective teams of engineers—not for each benchmark.

VIII. Discussion and Conclusions

We have demonstrated that an EDSL such as Accelerate is sufficiently platform-independent to break free of its original hardware target (CUDA/GPU) and create efficient programs on other architectures. This gives us hope that Harbb/Accelerate programs will be forward-portable to future parallel architectures and instruction sets.

The EDSL approach changes the playing field for the designer of compiler backends. Rather than contending with full blown languages and their complexities (e.g. pointers, aliasing, inheritance, virtual functions, etc), compiler backends can focus on simple value-oriented compute languages.

But the EDSL method solves only part of the performance-portability problem. Simple as EDSL target languages may be, there remains a substantial challenge in mapping them efficiently to the diversity of parallel architectures available now and in the near future. For example, the idiosyncrasies of memory bank access on NVIDIA GPUs must be taken into account to generate efficient implementations of the high-level collective operations that we have discussed.

This is a compiler backend research challenge. The *skeletons* method mentioned in Section III is one approach to this problem, as are the optimizations studied in the Copperhead [9] and Obsidian [20] projects. On the other hand, systems that rely on advanced optimizations typically suffer to some extent from performance-predictability problems. Thus achieving portable, predictable performance on a wide range of architectures—even for the simplest target languages—will be the subject of much future work.

References

[1] NVIDIA. (2009) NVIDIA's next generation cuda compute architecture: Fermi. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Architectural support for programming languages and operating systems*, ser. ASPLOS-IX. New York, NY, USA: ACM, 2000, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/378993.379006

[3] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. [Online]. Available: http://www.cse.unsw.edu.au/~chak/papers/CKLM+10.html

[4] Intel. Intel array building blocks. [Online]. Available: http://software.intel.com/sites/products/collateral/arbb/arbb_product_brief.pdf

[5] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, August 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360617

[6] M. I. Gordon *et al.*, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 151–162.

[7] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '08. New York, NY, USA: ACM, 2008, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1375657.1375675

[8] C. Elliott, "Programming graphics processors functionally," in *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004. [Online]. Available: http://conal.net/papers/Vertigo/

[9] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proc. of Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[10] Stanford. Stanford pervasive parallelism laboratory. [Online]. Available: http://ppl.stanford.edu/wiki/index.php/Pervasive_Parallelism_Laboratory

[11] M. M. Chakravarty, "C → Haskell, or Yet Another Interfacing Tool," in *In Koopman and Clack [23*. Springer-Verlag, 1999, pp. 131–148.

[12] M. Harris. Optimizing parallel reduction in CUDA. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[13] G. Mainland and G. Morrisett, "Nikola: Embedding Compiled GPU Functions in Haskell," in *Proceedings of the third ACM Haskell symposium on Haskell*. ACM, 2010, pp. 67–78. [Online]. Available: http://doi.acm.org/10.1145/1863523.1863533

[14] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate gpu vs. cpu performance without the answer," in *Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[15] Khronos OpenCL Working Group, "The opencl specification, version 1.0.29," http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf, 2008.

[16] M. Research. An introduction to microsoft accelerator v2. [Online]. Available: http://research.microsoft.com/en-us/projects/accelerator/accelerator_intro.docx

[17] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1353536.1346318

[18] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. of IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669121

[19] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[20] Joel Svensson, "Obsidian: GPU Kernel Programming in Haskell," Computer Science and Enginering, Chalmers University of Technology, Gothenburg, Tech. Rep. 77L, 2011, iSSN 1652-876X. [Online]. Available: http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=136543