Albatross: Systems Support for Augmented Reality

Christopher J. Rossbach[1] and Emmett Witchel[2]

[1]VMware Research *crossbach@wmware.com*
[2]The University of Texas at Austin *witchel@cs.utexas.edu*

Abstract

This position paper argues that future augmented reality (AR) and virtual reality (VR) applications require fundamental rethinking both of the structure of the software systems stack and of the front end programming tools available to developers for building such applications. Emerging hardware such as Google Glass, Oculus Rift, and Epson Moverio make the promise of immersive, interactive applications which incorporate the physical environment into both input and output seem tantalizingly close. However, a number of thorny research challenges must be addressed first. Such applications must produce substantially real-time results while relying heavily on compute-bound vision and learning applications as well high data-rate information produced by multiple sensors such as cameras. Such devices have limited compute and power budgets, necessitating offload for compute-intensive phases, potentially both to local specialized hardware accelerators or remote servers. However, the proliferation of vendor-specific platform variations forces the developer to avoid static assumptions about the presence or absence of particular accelerator resources. Consequently, the developer is faced with a compute fabric in which heterogeneity, distribution, and platform variability are the *de facto* standard, effectively barring the door for all but the expert programmer.

This paper presents our experience building a representative AR application for a representative currently shipping AR-amenable platform, and quantifies the performance and programmability challenges we encountered. Our experience suggests that the building distributed, interactive, AR applications requires a level of engineering effort that is infeasible if such applications are to become widespread in the future. We outline a system called Albatross, in which a reorganization of system components and programming tools have the potential to make such applications much easier to build in the future.

1 Introduction

A new class of hardware is enabling perceptual applications, which will become pervasive in the coming decade. Commodity devices like Google glass [24] provide unprecedented sensing and display capabilities for commodity devices. The recent proliferation of such devices presages the realization of an application domain long thought to border on science fiction: augmented reality (AR) applications, in which users interact with both device(s) and the real world under constraints set by real time and physical space. Such constraints make AR a distinct and strictly more challenging application space from virtual reality (VR), in which users interact interact only with device(s): VR attempts to immerse the user in a synthetic reality, while AR enhances the reality available to the user through the physical world alone.

Augmented reality applications have a huge spread of data and processing requirements. For example, detecting the presence of a human face-like object in a high resolution image requires a relatively small amount of processing of a small amount of data (e.g., a few megabytes), and has long been possible on mobile systems such as smart phones. By contrast, recognizing and *identifying* individual faces in a video stream requires significantly more resources and is a strong candidate for offload to specialized hardware such as image processors and GPUs. It is likely beyond the capacity of current mobile systems. At the extreme end of the spectrum, an entire data center might dedicate much of its resources to the collection of images and video, facial analysis, and the construction of facial models (e.g., recent developments at Facebook on robust support for facial recognition [23]). In all of these examples, the same processing tasks must be performed on radically different platforms: the *de facto* compute fabric for future AR applications features by necessity both heterogeneous and distributed resources.

Additionally, power and other design constraints result in a sea of devices and services which applications must manage themselves, creating the untenable situation where each application must reimplement the same

heuristics for managing data movement and computation for themselves. Multiple applications on the same platform must do their own management and resource discovery: there is no framework for orderly collection or dissemination of information relevant to all AR applications. AR applications need a new platform to coordinate the devices and services necessary to create rich perceptual applications. We propose a system called Albatross to address these needs. Albatross addresses challenges of heterogeneity, distribution, resource discovery, and data migration through a combination of language and runtime support.

We make the case for the necessity of Albatross by sharing our experience attempting to realize a representative AR application on modern infrastructure. To assess the realizability of distributed, heterogeneous AR applications on currently shipping hardware, we set a goal of building a simple application to run on a pair of smart glasses that would recognize the faces of researchers in our lab environment and label those researchers with their names in the field of view on the glasses. Our finding is that such an application is realizable, but only at a near herculean expense of non-reusable engineering effort. Our proposed system, Albatross represents the synthesis of our insights into the kind of development and runtime infrastructure that will be necessary to make such applications tractable for typical developers in the future. Albatross incorporates new abstractions, system structure, and programming tools for distributed heterogeneous AR applications.

2 Background and Motivation

In this section we provide a brief background on the augmented reality application domain, and motivate the need for Albatross with a case study: face recognition on smart glasses.

2.1 Background

The defining feature of augmented reality is the enhancement of real-world sensory input (e.g., a user's view of the world) with computer-generated sensory input such as graphics or sound. The conventional expectation is that the enhancement is performed in real-time, while maintaining coherence with environmental elements (e.g., a navigational aid application might project arrows indicating the user's next turn on the wind shield of a car.) AR applications fundamentally require sensors in addition to the processor, display and input devices that are typical of most computing platforms. The vast majority of today's mobile devices such as smart phones, tablets, and smart glasses, incorporate a rich array of sensors, including cameras, accelerometers, GPS, and so on, which in combination with their small form factor make them a tantalizingly suitable platform for AR.

However, while the defining features of AR can be crisply stated, the space of applications meeting the definition is dauntingly large: we further narrow the space to make the problem tractable. In this paper we primarily focus on AR applications whose user-facing artifacts run substantially on mobile devices, but which feature compute intensive and high data rate sensing, and for which interactive, or real time performance is required. This is still a large space: we hold that the vast majority of applications featuring computer vision and annotation of the user's view of the real world falls within this rubric. Moreover, mobile devices fitting this model are increasingly ubiquitous, and there is increasing demand for such applications.

Thus, we make the following assumptions about the subset of the AR application space we identify. First, the user-facing platform is mobile, and therefore assumed to be resource poor: compute cycles, power, and I/O are fundamentally limited. Local compute bandwidth is likely insufficient, necessitating offload to remote servers. Second, the application space features abundant data parallelism, primarily in the form of computer vision algorithms run on image streams captured from on-board cameras. Applications feature program phases that are a natural fit for offload to specialized hardware such as image processors, GPUs, FPGAs, and so on. A consequence of these assumptions is that common-case AR applications require offload, likely to both local and remote compute resources, which in turn may be heterogeneous. In short, a distributed, heterogeneous compute fabric is a fact of of life for the future AR developer.

Programming distributed systems is hard. Programming heterogeneous systems is hard. Programming embedded systems is hard. A dizzying array of programming tools and run-times, each requiring specialized expertise must be synthesized: this likely a fundamental frustration to the wide adoption of rich AR applications.

2.2 Case Study: Face Recognition on Smart Glasses

To motivate our proposed system, we explore a case study: face recognition on smart glasses. Note that face *recognition* and face *detection* are distinct. In face detection, the task is to return the absence or presence and location of a face or faces in an image. Any face will do. In face detection, the task is not just detect a face, but identify the party to whom the face belongs. Face recognition is a much harder problem, and incorporates face detection as just an initial phase. Our goal is to allow users to wear smart glasses; when other lab members come into the user's field of view, the application recognizes her face, and annotates the user's view by writing the lab member's name near that person in the field of view. We chose the workload not just because it is useful for researchers ill-equipped to remember each other's names: the workload is conceptually simple and involves well understood computer vision algorithms and components. Moreover, it is representative of a large class of computer vision based AR applications on mobile platforms: it is computationally demanding, incorporates real-time latency constraints (e.g., the system must annotate identified lab members before the user turns his head), and

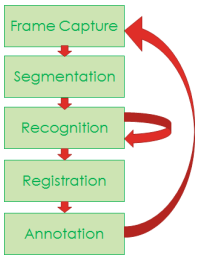

Figure 1: A face recognition system based on cameras

is rich with data-parallel algorithms. Intuitively, while it is a poor fit for compute-poor mobile devices, it is a plausible fit for remote offload, and many phases of the application are a natural fit for acceleration with specialized hardware.

Figure 1 shows a basic decomposition of a face recognition system. Components include:

frame capture: This component is responsible for capturing the raw image or video stream from the camera on board the smart glasses. Fundamentally, this component must run on the glasses.

segmentation: This component identifies segments of each image frame that are likely to contain an identifiable face. More directly, this component *detects* faces and narrows the region of interest (ROI) in the image frame to just those pixels containing potentially recognizable faces. This phase can be implemented in a CPU, GPU, or image accelerator hardware.

recognition: This component is responsible for the actual identification of faces. It examines ROIs from image frames to extract per-ROI feature vectors, which are then compared against a database of known users. The name of the closest fit is returned if that fit is within a pre-defined distance threshold. Otherwise, the face is deemed "unknown". This phase can be implemented in a CPU, or a combination of CPU and GPU hardware.

registration: This component is responsible for determining the mapping between the perspective of the camera and the perspective of the user to enable annotation of identified parties at the correct place in the field of view. Registration is a canonical element of AR applications, and is often the most latency and precision sensitive phase. While we could have invested more effort in better algorithms here, in our implementation, the registration is a simple triangulation based on some assumptions about typical physical displacement between the camera and the user's eyes. In this context, the simple approach is sufficient. Most candidate algorithms feature data parallelism that makes the phase a good fit for GPUs.

annotation: This is a simple rendering task. Registration returns a list per image frame of identities and locations (if any are found), and annotation is responsible for overlaying the name at the proper location in the user's field of view. While rendering can be offloaded to a remote server, its simplicity for our application meant our only logical choice was to annotate on the smart glasses themselves.

Figure 2: The Epson Moverio BT-200 Smart Glasses used in this study.

CPU	2 x ARM Cortex A9 (TI 4460 1.2 Ghz)
Memory	1GB DRAM
Storage	8GB flash
Camera	5MP photo/1080p video
Sensors	GPS, geomagnetic, accelerometer, gyroscope, optical touch
Network	802.11 b/g/n
OS	Android 4.04
Accelerators	IV3 (Codec), POWERVR SGX544 GPU image processor

Table 1: Epson Moverio BT-200 platform details.

2.3 Platform

We selected the Epson Moverio BT-200 [22] smart glasses as the user-facing platform. Figure 2 shows a picture of the glasses, and Table 1 details relevant platform parameters. Other competing smart glasses platforms are qualitatively similar. Obvious challenges arise from this (or any) smart glasses platform. First, the device is limited both in terms of compute bandwidth and I/O bandwidth. Local implementations are likely to be bottlenecked by local CPU, while remote offload is likely to be bottlenecked by transfer bandwidth and latency. Additionally, system structure plays a role: the Android platform features a deep system stack, with a fairly constrained, managed programming interface. The diversity of hardware supported by Android necessitates a programming model in which hardware diversity is abstracted away. Direct (read: "performant") access to native hardware resources is discouraged and requires significant additional engineering effort.

2.4 Different implementations

Naïve The naive implementation implements the entire application on the smart glasses device using a combination of Java (Android SDK v4.4) and JNI wrappers to invoke native implementations vision algorithms from the OpenCV (C++) library, version 2.49.

Basic offload The basic offload implementation offloads all phases of the application except frame capture and annotation (which arguably must run on the glasses) to a remote server. Segmentation, recognition, and registration are performed in native code on the server. There are tradeoffs in this part of the design space around which

	pixels	MB/s_{raw}	MB/s_{jpeg}
VGA	640x480	35.2	3.5
HD	1360x768	120	11.9
WQXGA	2560x1600	470	46.9

Table 2: Required data rates for various image sizes.

	Mb/s	MB/s
802.11n	150	18.8
802.11ac	866	108.3
802.11ad	6912	864

Table 3: Maximum data rates for various 802.11 standards.

protocols to use, and format for data transmission. Our finding was that the overheads incurred compressing the image capture stream (either to JPEG per frame, or as an H.264 compressed stream) introduced more latency than was eliminated by saving network bandwidth. The smart-glasses simply could not compress the images fast enough to enable a performance win. Consequently, the data shown are for the most performant variant in this design space, which transmits image frames in the camera's native NV21 format.

GPU offload The GPU offload implementation is the same as the basic offload implementation, but with segmentation and recognition performed on the GPU at the remote server. GPU algorithms were written using CUDA version 6.0.

Full-tilt boogie The full-tilt boogie implementation uses the image processor accelerator hardware to perform segmentation, the results of which are used to send only the image regions known to contain faces to the remote server. Recognition and registration are then performed on the GPU, and results are returned to the device which performs the annotation locally.

2.5 Performance

The data show that almost all of the implementations are unable to provide performance even close to usable. Implementing the face recognition application using only the tools provided by the Android SDK and NDK and only the compute resources available locally on the smart glasses device yields throughput and latency that render the application unusable, providing only 3 frames per second throughput at a latency that is readily perceptible and annoying to the user.

Simple offload to a remote server does not do much to improve the situation: indeed simple offload to a CPU-only remote server hurts performance, dropping the throughput to 2 frames per second due to a combination of network and CPU throughput problems at the server. Offload to a CPU+GPU server helps recover some of the lost performance by reducing the latency of the segmentation and recognition phases. Table 2 shows the data rate needs of the application for both raw and jpeg compressed images, while Table 3 shows the current and projected ideal data rates for current and future 802.11 standards. The tables suggest that future wireless network hardware may eventually support data rates that remove the network bottleneck for these implementations. We experimented with sending image data over the network in both raw and compressed formats, and found that the additional compute latency for compression was essentially equal to the gains resulting from using less network bandwidth for communicating images. This problem could be overcome using hardware acceleration for compression. We also note that H.264 is a non-starter for this application because the compression operates across frames. Time domain compression adds multiple image frames of additional latency at the receiving end, which translates to unacceptable impact on response time.

Finally, the full-tilt boogie implementation represents the only assignment of modules to compute resources that yields a usable system. By using hardware support for face detection on the cortex chip, we are able to reduce the latency of segmentation while at the same time reducing the consumption of network resources: only image regions containing faces are sent over the network, which translates to significant additional latency savings per frame. In combination with acceleration of the recognition phase on the GPU, the throughput and end-to-end latency of 60 ms per frame is able to essentially match the 30 frames per second rate produced by the on-board cameras.

While the performance characteristics of our different implementations suggest the full-tilt boogie implementation is clearly preferable, the complexity of the code and development effort tells a very different story. We hold that even the most naïve implementation demands a level of expertise that is beyond that of the typical developer, since using JNI was required to move segmentation and recognition phases into native code on the glasses. From the outset, familiarity with multiple languages and systems level issues around integration between managed and unmanaged code is required. The full-tilt boogie implementation makes use of Java, C/C++, CUDA, and requires dealing with specialized APIs and run-times for GPU, image accelerator, and network programming, as well as dealing with the laundry list of systems challenges that arise in any distributed systems development effort.

However, code complexity and developer expertise are not the worst problem in this picture. The real problem is the level of time and effort that went into searching the implementation space to find a mapping from task to resource that meets the performance requirements. Each assignment involves re-implementation of at least one component. Such brute force search of the implementation space is a fundamental barrier to the realization of this kind of application in the future. With Albatross we propose a framework that simplifies both the implementation of components across diverse execution targets, and automates the finding of a performant assignment.

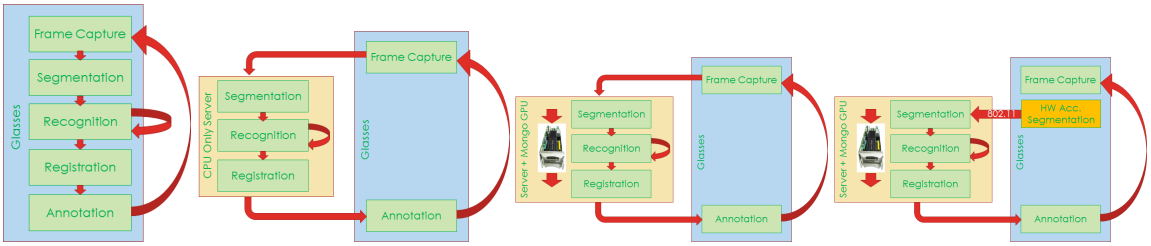

Figure 3: Different implementations of the face recognition application. From left to right: the naïve implementation, the basic offload implementation, the basic offload plus gpu system, and the realizable system.

implementation	client	server	fps	lat/frame	bottleneck
naïve	1000 Java + 500 C/C++	NA	3	350	CPU
basic offload	2200 Java	1000 C/C++	2	370	Wifi+CPU
basic offload	2200 Java	1000 C/C++ + 500 CUDA	6	180	Wifi
full-tilt boogie	3200 Java	1200 C/C++ + 500 CUDA	30	60	end-to-end latency

Table 4: Maximum data rates for various 802.11 standards.

3 Albatross

In Albatross, we propose the following restructuring of runtime components and front end programming tools.

Albatross features a front end compiler component similar to Delite [13], Copperhead [16] or Dandelion [43]. The front end compiler provides the developer much-needed insulation from heterogeneous programming challenges, by supporting a single-threaded, managed interface across diverse architectural targets. Each of the implementations we explored in the previous section required multiple languages, with the most performant requiring four. Albatross enables the programmer to express these applications in a single language. We plan to use an extended version of the Dandelion compiler to generate multiple implementations of every component in the data-flow graph of a given application. For the face recognition workload we explore here, the compiler would generate code in Java, C++, and CUDA for the segmentation and recognition tasks, as well as data movement and marshalling code to move image data across machines and across architectures. The back-door "hand-implementation" feature of Dandelion can be used to allow the user to point the compiler at already optimized implementations (e.g., OpenCV) for components when they are available.

Like Dandelion, StarPU [7], and PTask [42], Albatross enables the compiler to target a dataflow program structure. We use a distributed version of the PTask runtime as the compiler target for the front end, as well as the runtime which selects an optimal and potentially dynamic assignment of service providers to nodes in a dataflow graph, as described below.

The primary contribution of Albatross beyond synthesis and extension of previous work on heterogeneous and distributed frameworks is the introduction of a service brokering backplane which allows an application to enumerate available resources and acquire performance and latency estimates based on instrumentation of the network and remote service execution. At runtime, a particular application instance enumerates all services that export an interface matching a component in the dataflow graph created by the compiler. Because services can provide performance estimates, and the runtime estimates network bandwidths and latencies, graph algorithms can be used in combination with brute force search to find an optimal or at least acceptable assignment of application components to local or remote compute resources. To do this, the run-time or application builds a graph whose nodes weights are service latencies, and whose edges are communication latencies. As the application enumerates available applicable services, the edge and node weights of the graph can are varied accordingly. The application chooses the assignment of service providers to nodes that minimizes the critical path through the graph. Thus Albatross is able to automate the time- and engineering-brute force search of the implementation space, enabling future AR developers to deliver applications with drastically reduced effort.

Because Albatross may aggregate many services into a single application, and those services may represent a combination of many different providers, potentially in different administrative domains, Albatross needs a way to find relevant services across many providers. Albatross satisfies this need by providing a global query system; a query can in principle cover all services of all providers, access controls permitting. We propose a query language (a simplified form of SQL) to support this: each service is (logically) a database row, similar to the approach taken by Dremelčite. Queries in Albatross help applications compose services aggregated from many providers.

4 Related work

Albatross builds on ideas drawn from an array of different areas, including programming languages, compilers and run-times, operating systems, and distributed systems. In aggregate, the body of related work is significant; we touch briefly on the most relevant works here.

Scheduling and Deployment. Scheduling for heterogeneous systems is an active research area: systems

such as PTask [42], TimeGraph [32] and others [48] focus on eliminating destructive performance interference, while Maestro [45] but focuses on task decomposition, automatic data transfer, and auto-tuning of dynamic execution parameters. A large number of systems have taken on the challenge of sharing heterogeneous resources and dynamic assignment that affinitizes the best-fit architecture for a given task or computation phase [4, 5, 6, 8, 9, 10, 12, 19, 26, 31, 36, 44, 46], potentially in distributed environments [14, 18, 27, 30, 41, 50]. Albatross must address the same challenges and many techniques from these works will be directly applicable. However, none of these works address all the challenges of Albatross in aggregate; the problem of optimally dynamically remapping computations to utilize remote services implemented on potentially heterogeneous hardware remains an open problem.

We plan to build upon the service discovery and deployment techniques described by Dremel [37] and Sapphire [52]. These systems leave a number of unsolved challenges for Albatross, such as availability, privacy, and discovery and dynamic deployment in the presence of heterogeneous hardware.

Language-level support. StreamIt [47] and DirectShow [35], OmpSs [14], Hydra [49], PTask [42], and IDEA [20] all provide a graph-based dataflow programming models for offloading tasks across heterogeneous devices. Liquid Metal [29] and Lime [3] are programming platforms for heterogeneous targets. Flextream [28] is a compilation framework for synchronous dataflow models that dynamically adapts applications to FPGA, GPU, or CPU target architectures.

High-level language support for GPUs in high-level languages such as C++ [25], Java [2, 15, 39, 51], Matlab [1, 38], Python [17, 33], MapReduce [34], and .NET [43] is an active area of research, and much of this work will apply directly in the context of Albatross. Similarly, work on frameworks that leverage aspects of the problem domain to improve expressability or performance [11, 21, 40] may be re-targeted from a static setting to the dynamic setting addressed by Albatross.

5 Conclusion

This paper advocates a fundamental reorganization of system software and front end programming tools to meet the performance and programmability needs of future AR applications, which we argue must fundamentally run on distributed, heterogeneous compute fabric. We find that current commodity hardware can meet the bandwidth and latency needs of such applications, but at a staggering cost in engineering effort: better tools for programming this fabric are desperately needed. We propose Albatross, which decouples application structure and components from implementation, allowing a programmer to express applications in a single productivity language, while adapting its implementation dynamically to accommodate changes in topology, as well as available local and remote compute resources to find a performant partitioning of application work across those resources, with little or no effort from the programmer.

References

[1] Matlab plug-in for CUDA. `https://developer.nvidia.com/matlab-cuda`, 2007.

[2] JCuda: Java bindings for CUDA. `http://www.jcuda.org/jcuda/JCuda.html`, 2012.

[3] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, 2010.

[4] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*, Shangai, Chine, Dec. 2010.

[5] C. Augonnet and R. Namyst. StarPU: A Unified Runtime System for Heterogeneous Multi-core Architectures.

[6] C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE Architecture with the StarPU Unified Runtime System. In *SAMOS '09*, pages 329–339, 2009.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[8] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1:2003, 2003.

[10] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. 2004.

[11] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. In *SC 2006*.

[13] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. OLUKOTUN. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[14] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] P. Calvert. Part II dissertation, computer science tripos, university of cambridge, June 2010.

[16] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.

[17] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, 2011.

[18] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[19] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF 2008*, 2008.

[20] J. Currey, S. Baker, and C. J. Rossbach. Supporting iteration in a heterogeneous dataflow engine. In *SFMA*, 2013.

[21] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[22] Epson. *Moverio BT-200*, 2015.

[23] Facebook. *Facial Recognition App.*, 2015.

[24] Google. *Google Glass*, 2015.

[25] K. Gregory and A. Miller. *C++ Amp: Accelerated Massive Parallelism With Microsoft Visual C++*. Microsoft Press Series. Microsoft GmbH, 2012.

[26] D. Grewe and M. Oï£¡Boyle. A static task partitioning approach for heterogeneous systems using opencl. *Compiler Construction*, 6601:286–305, 2011.

[27] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, 2008.

[28] A. Hormati, Y. Choi, M. Kudlur, R. M. Rabbah, T. Mudge, and S. A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, pages 214–223, 2009.

[29] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/-software boundary. In *ECOOP*, pages 76–103, 2008.

[30] W. Jiang and G. Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. *Parallel and Distributed Processing Symposium, International*, 0:644–655, 2012.

[31] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC 2009*.

[32] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

[33] A. Kloeckner. pycuda. `https://pypi.python.org/pypi/pycuda`, 2012.

[34] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, Mar. 2008.

[35] M. Linetsky. *Programming Microsoft Directshow*. Wordware Publishing Inc., Plano, TX, USA, 2001.

[36] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[37] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[38] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, 2011.

[39] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using gpus from java. In *HPCC-ICESS*, pages 375–380, 2012.

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[41] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 140–147, 2012.

[42] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. Symposium on Operating Systems Principles (SOSP), October 2011.

[43] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. SOSP'13: The 24th ACM Symposium on Operating Systems Principles, November 2013.

[44] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP 2008*.

[45] K. Spafford, J. S. Meredith, and J. S. Vetter. Maestro: Data orchestration and tuning for opencl devices. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2010.

[46] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, 2012.

[47] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC 2002*.

[48] U. Verner, A. Schuster, and M. Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 120–129, New York, NY, USA, 2011. ACM.

[49] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *ASPLOS 2008*.

[50] P. Wittek and S. DaráNyi. Accelerating text mining workloads in a mapreduce-based distributed gpu environment. *J. Parallel Distrib. Comput.*, 73(2):198–206, Feb. 2013.

[51] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In *Euro-Par*, pages 887–899, 2009.

[52] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, Oct. 2014. USENIX Association.