Enabling Legacy Applications on Heterogeneous Platforms

Michela Becchi, Srihari Cadambi and Srimat Chakradhar
NEC Laboratories America, Inc.
{mbecchi, cadambi, chak}@nec-labs.com

Abstract

In this paper we make the case for a runtime technique to seamlessly execute legacy applications on heterogeneous platforms consisting of CPUs and accelerators. We consider discrete as well as integrated heterogeneous platforms. In the former, CPU and accelerators have different memory systems; in the latter, accelerators share physical memory with the CPU. Our proposed runtime does not require any code changes to be made to the application. It automatically schedules compute-intensive routines found in legacy code on suitable computing resources, while reducing data transfer overhead between the CPU and accelerators. To reduce data movement, our runtime defers data transfers between different memory systems, and attempts to move computations to data instead of vice-versa. This could create multiple copies of the data – one on the CPU, and the others on the accelerators - leading to coherence issues. To address this problem, we propose adding an operating system module that maintains coherence by intercepting accesses to shared data and forcing synchronization. Thus, by exploiting existing mechanisms found in system software, we architect a non-intrusive technique to enable legacy applications take advantage of heterogeneous platforms. With neither software changes nor additional hardware support, the proposed system provides a unified compute and memory view to the application programmer.

1. Introduction

For many commercial and consumer workloads with scientific, media-rich and graphically intense portions, heterogeneous platforms strike a balance between performance, energy and development cost. One form of heterogeneity is represented by platforms with one or more multi-cores (e.g., x86 CPUs) coupled with many-core processors (e.g., GPUs) and/or other accelerators (FPGAs, cryptographic processors, etc.). The CPU usually controls the offloading of computations to different accelerators.

Library-based programming eases the burden of deploying applications on heterogeneous systems. In this scenario, applications invoke well-known computational routines that are made available in pre-compiled libraries. For some of these, multiple library implementations targeting different computational units are provided. A notable example is dense matrix multiplication, implemented in the *sgemm* [14] and *cublasSgemm* [15] libraries, targeting x86 CPUs and Nvidia GPUs respectively. In recent years there has been a lot of activity in porting and accelerating computational routines on GPUs, and we do not expect this trend to change in the near future. Selection mechanisms to choose the execution unit for given computational kernels based on their estimated performances are described in several recent research efforts [2][3][8].

In this work we make the case for a runtime to increase the performance of existing applications on heterogeneous platforms *without requiring any code changes*. In particular, legacy applications written using library APIs can be transparently accelerated by intercepting library calls, and invoking a suitable accelerator implementation.

We believe that a runtime that automatically selects accelerators for legacy code should also take into account data transfers, especially because large transfers over conduits such as PCI can overwhelm any speedup achieved by the accelerator. In [16] we proposed a data-aware runtime to efficiently run applications on heterogeneous systems. After an accelerator processes a function, the runtime *defers* transferring the function's data back to the CPU until the data are required. Such an on-demand transfer policy enables the runtime to move computations to the data, rather than the other way around. With the data-aware runtime on real applications, we measured speedups of 25% over a data-agnostic runtime.

Such an on-demand transfer policy creates multiple copies of data and necessitates source-level changes (such as locks and synchronization points) or hardware support in order to maintain data coherence. However, our goal is to address *legacy* applications, that is, to seamlessly port them and minimize data transfer with no code changes. To this end, in this work we propose using the operating system and runtime to provide the programmer with a unified memory view of possibly discrete underlying memory sub-systems. Besides scheduling computations and managing data movement between CPU and accelerators, the runtime must *ensure coherence of data present in multiple locations* without source code or hardware changes.

The idea of minimizing the overhead due to data transfers and of moving computation to data has been previously considered in the context of programming models for heterogeneous platforms [5][6][7][8][9]. However, our requirement of addressing legacy applications introduces significant distinctive challenges. First, our design is driven by the need for avoiding any code changes in the application. On the contrary, programming models as those listed above require the application to be

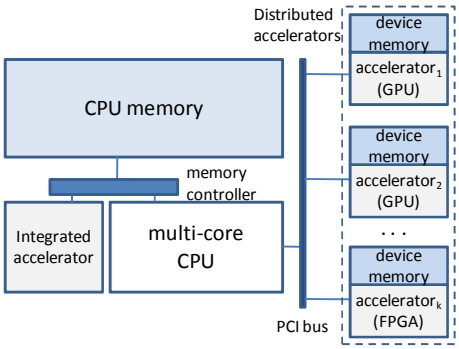

Figure 1: Target heterogeneous platform.

written according to given API. Second, we assume that data access patterns are not known a priori and cannot be predicted. Therefore, memory access optimizations and synchronizations must be performed online with no knowledge of future programs behaviors.

We believe that a comprehensive solution should consider two kinds of devices: "distributed" and "integrated" accelerators (Figure 1). Distributed devices have their own local memory, and are typically connected to the CPU via a PCI-bus. Integrated accelerators share physical memory with the CPU; an example is the Ion platform [17], where an Nvidia GPU shares memory with an Intel Atom CPU. As we will discuss in Section 4, different memory organizations lead to different design issues, both in terms of memory usage patterns and required mechanisms for data coherence.

We recognize the existence of obvious similarities between the memory models we consider in the context of heterogeneous platforms and the traditional distributed [6] and shared [5] memory models adopted in the context of (homogeneous) multi-core architectures. However, the presence of many-core accelerators introduces some interesting design questions. First, since GPU (and most accelerators) do not run an operating system, they do not offer inherent mechanisms to trap memory accesses. Such mechanisms are typically needed to synchronize data accesses across different memories. Tracking of all accesses to the accelerator memories must therefore be implemented in the runtime system, and this can be efficiently done only in a coarse grained fashion (e.g., on transfers of entire function call parameters). Second, since the accelerator processing time can be order of magnitude smaller than the CPU processing time, the synchronization overheads can weigh differently (and in a more significant way) compared to the homogeneous case.

Other related research efforts merit discussion. Harmony [2] is a runtime that schedules functions on heterogeneous systems taking input size into account. Qilin [3] presents an adaptive mapping technique to split and concurrently run a function across a CPU and a GPU with the goal of improving performance. These two proposals, however, focus on the compute aspect and do not optimize memory transfers. StarPU unified runtime system [11] proposes implementing CPU-GPU memory coherence using the MSI protocol. However, it requires programmers to rewrite their application using a new API. The authors of [8] propose tools to encapsulate different processor-specific tool-chains and language mechanisms in order to enable applications on heterogeneous systems. In particular, they consider using the Merge [9] framework to optimize data transfers. This mechanism, however, does require analysis and modification of the application source code. SEJITS [10] addresses programmability of heterogeneous platforms by proposing the integration of components written using productivity and efficiency programming languages, whereas [4] discusses a programming model for heterogeneous x86 platforms. However, these two proposals do not target legacy applications. Finally, GViM [12] is a Xen-based virtualization framework for heterogeneous systems that reduces the number of user space to kernel space copies when moving data between the CPU and GPU. However, this does not optimize the data transfers between CPU and GPU. In summary, our proposal offers a different viewpoint by focusing on compute and memory unification in the context of legacy applications.

The rest of the paper is organized as follows. In Section 2, we use a simple example to illustrate the benefits of data-aware scheduling. In Section 3, we give an overview of the proposed runtime system. In Section 4, we provide more details on our solution to the memory unification problem. We conclude in Section 5.

2. The Case for Data-Aware Scheduling

In this section, we use a real application, Supervised Semantic Indexing (SSI) matching [13], to illustrate the performance potential of data-aware scheduling on heterogeneous systems. SSI is an algorithm used to semantically search large document databases. It ranks documents based on their semantic similarity to text-based queries. Each document and query is represented by a vector, with each vector element corresponding to a word.

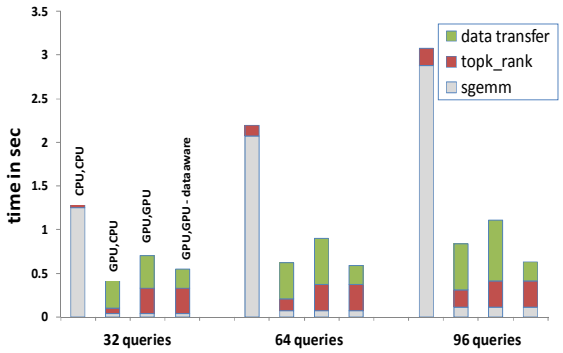

Figure 2: SSI matching performance on a discrete heterogeneous system.

We omit further algorithmic details in the interest of space, and refer interested readers to [13].

The SSI matching process has two compute-intensive functions. The first *(sgemm)* is a dense matrix multiplication of the query vectors with all document vectors. The second routine *(topk_rank)* must select, for each query, the top *k* best matching documents. With millions of documents to search for each query, these two functions take up 99% of the SSI execution time.

We perform four runs of SSI matching, each with a different schedule for the two functions. For each run, we consider 32, 64 and 96 simultaneous queries into a 1.6M document database, identifying 64 best matching documents for every query. The document database contains documents selected from the Wikipedia [13]. For matrix multiplication, we use the Intel MKL [14] on the CPU and the CUBLAS library [15] on the GPU. We use our custom CPU and GPU implementations for *topk_rank.*

Figure 2 shows the total running time reported on a heterogeneous platform consisting of a Xeon 2.5GHz quad-core CPU with an nVIDIA Tesla C870 128-core GPU. For each number of parallel queries, we evaluate the following schedules: (i) both functions on CPU, (ii) *sgemm* on GPU, *topk_rank* on CPU, (iii) both functions on GPU and (iv) both functions on GPU with data transfer deferring (i.e., data-aware scheduling). While a significant time reduction is seen from porting *sgemm* to the GPU, we note that *topk_rank* is actually faster on the CPU. We also note that data transfer takes up at least half the run time, and that its contribution to the total runtime increases with the input size. Using data-aware scheduling, the data transfer after *sgemm* is deferred until the runtime sees the next function (*topk_rank*). At this point, the runtime weighs the options of moving *topk_rank* to the slower GPU, or moving the data back to the CPU. Figure 2 shows that, for large inputs, the former choice results in about a 25% speedup.

3. System Overview

We now describe what a system that provides a unified compute and memory view to legacy applications on heterogeneous platforms should include. The main components of the system (Figure 3) are *function libraries* and a *runtime*. The runtime itself consists of a *library call module* and an OS *memory unification module*. Different implementations of well-known function libraries targeting CPU and various devices (GPUs, accelerators, etc.) are provided, possibly by third-parties. Calls to these library functions are intercepted, and the runtime determines the implementation to instantiate and the computational unit to use. This decision depends not only on function execution time and computational unit availability, but also on the estimated data transfer overhead. Minimizing this overhead could be achieved by avoiding useless data transfers and, whenever possible and desirable, by moving computations to the data. Since the (legacy) application is not known a

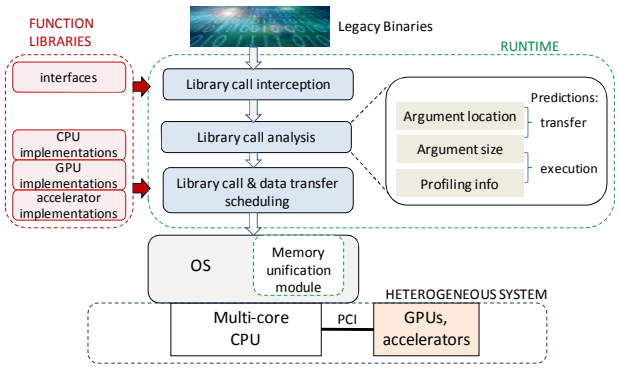

Figure 3: Overall System: Runtime and OS Module.

priori, these decisions must be made dynamically. If minimizing data transfers creates multiple copies of the data, the OS memory unification module assists the runtime in keeping the copies coherent.

We assume that function library implementations are "black boxes" to the runtime, whereas the library API is exposed. Thus, the only data transfers that the runtime can optimize correspond to the API function arguments. Further, data transfers between CPU and device memory can be triggered only by the runtime. To this end, we make three assumptions. First, function library implementations operate on the memory of the target device. The GPU implementation of *sgemm*, for instance, will assume that the matrices pointed to by the function arguments reside on GPU memory. Second, for each pointer argument, the function library interface must be annotated with the following information: (i) whether the corresponding parameter is read-only, write-only or read-write from the function's perspective and (ii) the size of the data structure the argument points to. The *sgemm* interface, for instance, will be annotated as follows:

```
void sgemm(char transa, char transb,
           int m, int  n,  int  k,
           float alpha, float *a, int lda,
           float *b, int ldb, float beta,
           float *c, int ldc);

Arg 'a': read-only :: (transa='n') ?
          (lda * k) : (lda * n)
Arg 'b': read-only :: (transb='n') ?
          (ldb * n) : (ldb * k)
Arg 'c': read-write :: (ldc * n)
```

This annotation allows automatic generation of the code to intercept library calls and invoke data transfers (more details are provided in Section 4.2). In addition, for each device type in use, the runtime must be provided with primitives to allocate device memory and transfer data between CPU and device memory. In the case of integrated devices, the runtime must also be provided with primitives to allocate page-locked host memory to those devices. For GPU devices, for instance, CUDA's `cudaMalloc`,

`cudaMemcpy` and `cudaHostAlloc` primitives can be used for this purpose.

The library call module must intercept library calls, analyze argument size and location, estimate data transfer and execution time on the available computational units, and redirect calls to the most suitable unit after having triggered necessary data transfers. We assume that each library implementation has been profiled on the available computational units for different input sizes. The gathered profile information, along with the actual arguments, can be used to estimate execution time. The data transfer time depends on the size and location of the function call parameters. The location information can be provided by the memory unification module. Note that, as far as execution time estimation is concerned, mechanisms proposed in related work [2][3][8] can be adopted. In particular, an additional requirement should be considered when using GPU-CPU work splitting [3]: the function library interface should be annotated with a mechanism for splitting computations into sub-computations and merging intermediate results. This can be done by using a Sequoia-like [1] syntax.

4. Memory Unification: Our Proposal

The memory unification module provides a homogeneous view of the memory system, optimizes data transfers, and ensures coherence across the different memory modules. When targeting heterogeneous systems such as those in Figure 1, these requirements lead to a set of design issues.

4.1 Design Issues

The first issue is *data coherence*. In a traditional, distributed accelerator-based system, the input arguments of a function call are copied to device memory before invocation, and the outputs are transferred back to host memory afterwards. This can trigger unnecessary data transfers, especially between multiple function calls that are invoked in sequence on the same device. Prior work [9] addressed this by defining new library functions obtained by composing existing ones. This, however, leads to the need for an *a priori* application analysis, and optimizes only a class of data transfers. In our view, data transfers between different memory elements should be triggered by the runtime only *on demand*. At any given time a data structure may reside on multiple memory elements, and not all the copies may be up-to-date. In this situation, the runtime must ensure that every access is coherent.

A second design issue is whether data replication on more than one accelerator should be allowed. Being the "master" unit, a (possibly outdated) copy of the data will always reside on CPU memory. Copying data between two accelerator memories involves an intermediate copy on the CPU memory. If an application consists of a sequence of library calls, and one library call is scheduled on a device, it is likely that the next call will be scheduled on the same device. We believe that allowing data to reside in parallel on multiple devices would complicate coherence handling without performance pay-offs. Therefore, we opt to limit data replication to a single accelerator memory.

A third design issue pertains to the use of two kinds of devices: "distributed" and "integrated". In the former case, the device has its own memory and data transfers between CPU and device memory are required. In the second, shared address spaces on CPU memory can be created by using page-locked memory, mapping it into the device space and "relocating" the shared data into it. Note that if not enough page-locked memory is available, the CPU and the integrated device will access separate memory regions on the same physical memory, and the coherency problem is addressed as in the distributed case.

4.2 Design Direction

Here we describe a possible implementation of the proposed runtime. In order to control data transfers and handle data coherence, the memory unification module must maintain a mapping between CPU and device memory regions and provide an API to query the binding between different memory spaces, obtain the location of data structures, and perform device memory allocation, de-allocation and data transfers. This API can be invoked by the library call module when intercepting function calls.

However, if data are distributed across different memories and data transfers are deferred, synchronizing at the library call granularity is not sufficient to guarantee data coherence. Data accesses happening outside the intercepted function calls would be unsynchronized, leading to possibly incorrect operation. To address this, we integrate the memory unification module within the operating system. The idea is *to handle data synchronizations outside library function calls by forcing page faults and having the page fault exception handler invoke the memory unification module.* Thus, the memory unification module API can be invoked by two entities: the library call module and the page fault exception handler. We now describe a design under the Linux operating system.

Linux associates each running process with a list of memory regions each assigned a set of access rights and a set of virtual addresses [18]. Similarly, our memory unification module associates each process with a list of non-overlapping *data blocks*, each one representing virtual address regions that have been mapped onto a device. Each data block may cover a subset of a memory region or may span across several memory regions.

Each data block, shown in the top part of Figure 4, consists of a pointer to the CPU's memory region, a device address where the memory contents were transferred, its size, a location (identifier of the device in case of multiple devices) and a synchronization status, indicating whether the up-to-date copy of the data in the block resides in CPU or device memory. Additionally, in the case of integrated

devices, an additional field indicates the address in page-locked memory where the data block has been relocated.

Data block creations are invoked by the library call module. A new data block is instantiated when a virtual address range is first accessed by a device implementation of a library function. Data synchronizations outside library function calls are forced by manipulating the page table entries of the interested memory regions and extending the page fault exception handler. Data block creation and CPU data access handling are performed as described below.

If the function call is scheduled on a PCI-connected device, then the access rights of the function call arguments are important[1]. If the argument is read-only, then device memory is allocated and data are initially synchronized by performing a host-to-device memory transfer. To handle coherence, all OS page table entries corresponding to the given address range are marked as read-only. Any subsequent read access to the data block will be allowed, whereas any write access will trigger a page fault. Note that a write access implies that the CPU code is modifying a data structure which has been copied to a device. Therefore, in this situation the page fault handler will resolve the fault by setting the synchronization status of the data block to "up-to-date on CPU". Any subsequent device access to the data block will trigger synchronization. Note that this mechanism defers data transfer.

If the argument is write-only, then device memory is allocated but no data transfer is initially required (in fact, the data block is supposed to be written by the function call that executes on device memory). All OS page table entries corresponding to the given address range are marked as invalid. In this case, any subsequent CPU access to the data block will trigger a page fault. Faults caused by read operations will be resolved into device-to-host memory transfers unless the data block is already in synchronized status. Faults caused by write operations will be resolved by setting the data block synchronization status to "up-to-date on CPU."

If the argument is read-write, then device memory is allocated and data synchronized by performing a host-to-device memory transfer. All OS page table entries corresponding to the given address range are marked invalid. Page fault handling is similar to the write-only case.

If the function call is scheduled on an integrated device, then the system tries to allocate page-locked memory. If this operation is not successful, then the data block handling described in the distributed case will be performed. Otherwise, data will be relocated to the newly allocated region, which is shared between CPU and device. To ensure coherence, any subsequent CPU access to the virtual addresses in the data block should be redirected to the shared area. This is accomplished by marking all OS

[1] Note that arguments access rights are considered from the library function perspective.

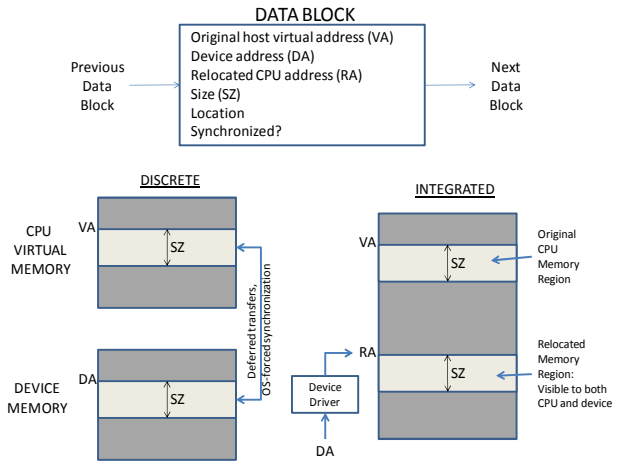

Figure 4: Data block (top) and CPU, device memory regions (bottom).

page table entries corresponding to the virtual address ranges in the block as invalid. The page fault handler will resolve the fault by redirecting the access to the shared area. After the initial copy, no additional data transfer is required.

The operation of the resulting page fault handler and its interactions with the memory unification module are summarized in Figure 5. Note that no page faults are triggered in case of read accesses to read-only arguments.

We consider the structure of the library call module to illustrate its interactions with the memory unification module. For each library `func` having (read-only) input parameters `r_pars` and (write-only) output parameters `w_pars`, the module contains a function whose structure is exemplified in the pseudo-code below. Handling integrated devices is omitted here for brevity.

```
(1)  void func(r_pars, *w_pars){
(2)    target = eval_target(&func,r_pars);
(3)    if(target==CPU){
(4)      cpu_func(r_pars,w_pars);
(5)      for (p in w_pars) mum->touch(p);
(6)    }else{
(7)      r_pars_d = w_pars_d = Ø;
(8)      for (p in r_pars)
(9)        r_pars_d U= mum->get(target,p,T);
(10)     for (p in w_pars)
(11)       w_pars_d U= mum->get(target,p,F);
(12)     dev_func(r_pars_d, &w_pars_d);
(13)     for (p in w_pars) mam->set(p);
(14)   }
(15) }
```

The `cpu_func` and `dev_func` routines represent the CPU and device implementation of the intercepted function, whereas the `mum` object represents the memory unification module API. The `eval_target` routine evaluates the target computational element based on size and location of the input parameters and on profiling information.

If the `eval_target` routine establishes that the execution must happen on the CPU (lines 4-5), then

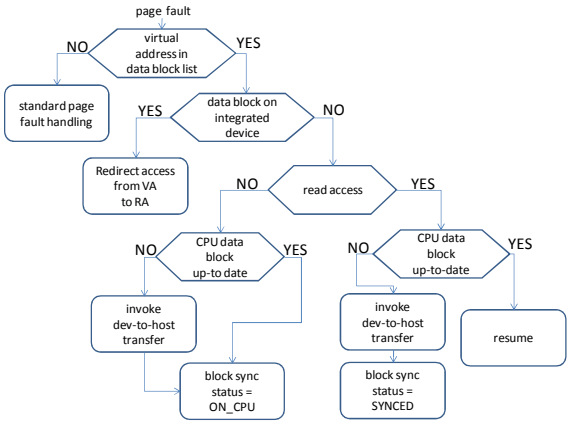

Figure 5: Page fault handler flow diagram.

`cpu_func` must be invoked. After execution, the `touch` primitive marks the output parameters as residing on the CPU memory. This operation does not imply any immediate data transfer.

If the function execution must take place on the device (lines 7-13), then `dev_func` is invoked. However, this operates on device memory. Therefore, a local copy of all input and output parameters (`r_pars_d` and `w_pars_d`) must be created (lines 7-11). For each parameter, the `get` function returns the pointer to that copy (and, if necessary, instantiates a new data block, allocates the corresponding memory on device and performs data synchronization). The last parameter of the `get` call specifies whether the device must have an up-to-date copy of the data, which is necessary only for the input parameters. After function execution, the output parameters are marked as residing on the GPU by the `set` primitive (line 13). Again, this does not imply any data transfer.

Data blocks can be resized or merged during execution: the interested reader can refer to [16] for a detailed description of data resizing and merging scenarios. Data block de-allocation (and device memory de-allocation) is performed in two situations: when a process terminates, and when device memory gets full. In this case, a LRU policy can be used to determine which data blocks to de-allocate.

5. Conclusion

In this paper, we make the case for a runtime system to seamlessly execute legacy applications on heterogeneous nodes consisting of CPUs and accelerators, thus hiding the underlying heterogeneity in terms of computing and memory elements. The runtime intercepts function calls to well known libraries, and schedules them on the appropriate computing resource after having analyzed the arguments and determined the location of the corresponding data. The overhead due to memory transfers is minimized by moving computations to data and deferring memory transfers until required by data accesses. Data coherence is ensured by extending the operating system with a memory unification module. With neither software changes nor additional hardware support, the proposed system provides a unified compute and memory view to the programmer.

References

[1] K. Fatahalian *et al*, "Sequoia: Programming the memory hierarchy," in Proc. of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL.

[2] G. Diamos and S. Yalamanchili. "Harmony: an execution model and runtime for heterogeneous many core systems," in Proc. of HPDC 2008, New York, NY.

[3] C. Luk, S. Hong and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in Proc. of MICRO 2009, New York, NY.

[4] B. Saha *et al*, "Programming model for a heterogeneous x86 platform," in Proc. of PLDI 2009, Dublin, Ireland.

[5] OpenMP: http://openmp.org/wp/

[6] A. Basumallik, S. Min and R. Eigenmann, "Programming Distributed Memory Sytems Using OpenMP," in Proc. of HIPS'07.

[7] "Writing Applications for the GPU Using the RapidMind™ Development Platform", http://tinyurl.com/rapidmind.

[8] M. D. Linderman *et al*, "Embracing Heterogeneity – Parallel Programming for Changing Hardware," in Proc. of HotPAR'09, Berkeley, CA, March 2009.

[9] M. D. Linderman *et al*, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in Proc. of ASPLOS 2008, March 2008.

[10] Bryan Catanzaro *et al*, "SEJITS: Getting Productivity And Performance With Selective, Just-In-Time Specialization," in Proc. of PMEA'09, Raleigh, NC, Sept. 2009.

[11] Cédric Augonnet *et al*, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in Proc. of Euro-Par 2009, Delft, The Netherlands, August 2009.

[12] V. Gupta *et al*, "GViM: GPU-accelerated virtual machines," in Proc. of HPCVirt '09, New York, NY, 2009.

[13] B. Bai *et al*, "Learning to Rank with (a lot of) word features," in Special Issue: Learning to Rank for Information Retrieval. Information Retrieval. 2009.

[14] MKL Library: http://software.intel.com/en-us/intel-mkl/.

[15] CuBLAS Library: http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf.

[16] M. Becchi *et al*, "Data-Aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory", in Proc. of SPAA 2010, June 2010.

[17] Zotac Ion board (Tom's Hardware review): http://www.tomshardware.com/reviews/zotac-ion-atom,2300.html.

[18] D. Bovet and M. Cesati, "Understanding the Linux Kernel," 3rd edition, O'Reilly.