Data-Aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory

Michela Becchi, Surendra Byna, Srihari Cadambi and Srimat Chakradhar
NEC Laboratories America, Inc.
4 Independence Way, Princeton NJ 08540
{mbecchi, sbyna, cadambi, chak}@nec-labs.com

ABSTRACT

In this paper, we describe a runtime to automatically enhance the performance of applications running on heterogeneous platforms consisting of a multi-core (CPU) and a throughput-oriented many-core (GPU). The CPU and GPU are connected by a non-coherent interconnect such as PCI-E, and as such do not have shared memory. Heterogeneous platforms available today such as [9] are of this type. Our goal is to enable the programmer to seamlessly use such a system without rewriting the application and with minimal knowledge of the underlying architectural details. Assuming that applications perform function calls to computational kernels with available CPU and GPU implementations, our runtime achieves this goal by automatically scheduling the kernels and managing data placement. In particular, it intercepts function calls to well-known computational kernels and schedules them on CPU or GPU based on their argument size and location. To improve performance, it defers all data transfers between the CPU and the GPU until necessary. By managing data placement transparently to the programmer, it provides a unified memory view despite the underlying separate memory sub-systems.

We experimentally evaluate our runtime on a heterogeneous platform consisting of a 2.5GHz quad-core Xeon CPU and an NVIDIA C870 GPU. Given array sorting, parallel reduction, dense and sparse matrix operations and ranking as computational kernels, we use our runtime to automatically retarget SSI [25], K-means [32] and two synthetic applications to the above platform with no code changes. We find that, in most cases, performance improves if the computation is moved to the data, and not vice-versa. For instance, even if a particular instance of a kernel is slower on the GPU than on the CPU, the overall application may be faster if the kernel is scheduled on the GPU anyway, especially if the kernel data is already located on the GPU memory due to prior decisions. Our results show that data-aware CPU/GPU scheduling improves performance by up to 25% over the best data-agnostic scheduling on the same platform.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *run-time environments.*

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

General Terms

Performance, Design, Experimentation.

Keywords

Heterogeneous platforms, multi-core processors, accelerators, distributed memory, runtime.

1. INTRODUCTION

Heterogeneous platforms consist of one or more multi-core general-purpose CPUs and one or more throughput-oriented many-core processors (for example, GPUs). Driven by many emerging data-parallel applications and by the need for higher performance, server vendors are beginning to rapidly commercialize these platforms [9], a trend that is expected to continue.

The most straightforward (and currently available) configuration that allows fast time-to-market is to have the many-core processor on an add-on card that is connected to the system via a non-coherent interconnect such as PCI Express. This has two implications. First, the CPU processing and its memory sub-system is completely separated from the many-core processor (GPU) and its memory sub-system, *i.e.*, the two memory sub-systems are not coherent and there is no shared memory. This makes programming difficult since the programmer has to manage the data that is manipulated by the CPU as well as the GPU. Second, with current PCI-E bandwidths, large data transfers between the two processing sub-systems can at times overwhelm any speedup achieved by the many-core when the processing alone is taken into consideration.

Such "loosely-coupled" distributed memory heterogeneous systems do not present the programmer with a unified view of the memory and compute elements. Today, applications that require acceleration from heterogeneous systems must be carefully profiled to discover data-parallel portions ("kernels") that could benefit from a many-core GPU. Once GPU custom implementations for those kernels are available, the application developer must explicitly schedule not only the kernel computations but also the required data transfers.

Ideally, a heterogeneous system should enable any legacy code written for homogeneous systems to run faster, in a way that is transparent to the programmer. GPU libraries for commonly available kernels (such as linear algebra for example) are necessary in order to enable this, but are not enough to allow complete transparency. A runtime that schedules computations as well as data transfers in order to maximize performance is required.

In this paper, we propose such a runtime. As a significant difference from past work [3], our runtime is cognizant of data transfer overheads and dynamically schedules operations taking

into account not only the predicted processing performance, but also data transfers. For instance, suppose an application has three candidate kernels with both CPU and GPU implementations. Assume that during a certain execution path, the first kernel is estimated to be much faster, but the second and third much slower on the GPU (based, say on the sizes of their parameters). Given this information, a data-agnostic scheduler is likely to run the first kernel on the GPU, transfer data back and run the remaining two kernels on the CPU. However if the first kernel produces a large amount of data that is consumed by the second kernel, a better schedule may be to run the second kernel also on the GPU and avoid the intermediate data transfer. With legacy code, the system is unaware what will follow the first kernel. After running the first kernel on the GPU, our runtime postpones data transfers back to the CPU until necessary. This way, when the second kernel is encountered, it can make a more informed decision taking into account the data transfer overhead, as well as the estimated performance. Although the GPU is slower in processing the second kernel compared to the CPU, running the kernel on the GPU could still result in an overall speedup. Our runtime analyzes these situations using simple, history-based models to predict processing as well as data transfer time, and uses these to guide the scheduling policy. It intercepts calls to candidate kernels, examines their arguments, and uses historical information and prior decisions to devise a schedule on-the-fly. Once a decision is reached for a kernel, the runtime invokes its CPU or GPU implementation transparently to the user.

One basic objective of our runtime is to target legacy code. In other words, we do not require source code modifications, but assume that the application performs function calls to well known kernels for which implementations targeting both the CPU and GPU are available. Our runtime's invocation happens by function call interception. This mechanism alone would lead to data coherence problems when accesses performed outside function calls target data residing on the GPU. In [7], we discuss operating system modifications to avoid this problem. In particular, the proposed design adds synchronization points within the page fault handler; at each synchronization point our runtime's API is invoked. As an alternative, it is possible to add synchronization points in the application via minimal source-level annotation (we discuss this in Section 4). Note that our goal is to propose a runtime, therefore we do not force the user to code a new application (or recode an existing one) according to specific primitives or frameworks. Therefore, our work fundamentally differs from previous efforts that propose programming models [1][5].

In summary, our contribution could be viewed as a runtime for a heterogeneous platform that provides a unified memory and compute view to the programmer despite the underlying platform being composed of two separate CPU and GPU, each one having its own memory sub-system. Such a view is enabled by automatically scheduling kernels on the compute units while simultaneously optimizing data placement on the two memories. The goal of the runtime is to maximize the overall application performance without requiring application rewriting. In [7] we discuss how the design can be generalized to the case where multiple GPUs or accelerators are connected to the CPU, and to that where CPU and GPU share (pinned) memory regions residing on the CPU.

The rest of the document is organized as follows. In Section 2, we overview closely related work. In Section 3, we present a motivational example that illustrates the benefits of data-aware scheduling. In Section 4, we detail the design, implementation and operation of our proposed runtime. In Section 5, we present an experimental evaluation on some real and synthetic applications. We conclude in Section 6.

2. RELATED WORK

Various programming languages and libraries have been introduced by multi-core CPU and many-core GPU vendors to utilize their computing power. Nvidia's CUDA [8], AMD's Brook+ [10], Intel's TBB [11] and Ct [12] provide programming interfaces to better utilize the underlying hardware. These interfaces, while making programming easier, target specific GPUs or CPUs and not a heterogeneous platform containing both. Similarly, solutions such as Microsoft Accelerator [13], Rapidmind [14], and Google's Peakstream [15] only target GPUs. Programming models targeting heterogeneous and distributed memory systems are presented in [1] and [5]. However, programming models and languages can be used either to write new applications, or to rewrite existing ones. In contrast, this proposal aims to enable legacy applications on heterogeneous platforms without requiring source code modifications.

PGI Accelerator [16], CAPS HMPP workbench [17], and HPC Project's Par4All [18] developed compilers to generate CUDA code for data parallel portions, especially loops. These tools use compiler level information to decide whether to run a code segment on CPUs or GPUs, but do not take the cost of data transfers into account. Compiler solutions also cannot make scheduling decisions that are best made at runtime. Our runtime uses parameters such as data size and data locality and decides when to schedule computations as well as data transfers.

Multicore-CUDA (MCUDA) [19] and Ocelot [20] translate CUDA code to run on multi-core processors. Liao *et al* [21] propose a similar translation of Brook-like code into multithreaded CPU code. The goal of these approaches is to provide a way to develop code for both CPUs and GPUs, but scheduling them on the hardware is left to the programmers.

OpenCL [22] attempts to provide a common programming interface for multi-core and many-core platforms. However, the programmer must decide the mapping of the kernels to the processing elements. IBM's OpenMP for Cell [23] and Intel's Merge framework [24] are also capable of running code on both CPUs and GPUs, but the mapping is not automatic (and involves code modifications). Harmony [3] proposes a runtime to schedule kernels either on CPU or on GPU based on estimated kernel performance using an API. Qilin [4] proposes an API with automatic and adaptive mapping support, which reduces the decision burden on programmers. However, Qilin's mapping is based on a curve fitting model to split computation on CPU and GPU. Neither Harmony nor Qilin consider the data transfer overhead, especially for legacy code. StarPU's unified runtime system [34][35] proposes implementing CPU-GPU memory coherence using the MSI protocol. However, it requires programmers to use a new API proposed by the system.

Data-aware scheduling strategies exist in cluster and distributed computing [28][29][30]. However, these techniques are used in scheduling jobs to fit data in disks and to reduce data transfers among cluster and grid computing nodes. We focus on a node within a cluster.

Table 1: Processing and data transfer times for SSI classification. For each input data size, the schedule resulting in the best performance is highlighted.

Number of simultaneous queries (*Q*)	Kernel Location (Schedule)		Kernel Processing Time			Data Transfer Time	Overall Speed
	sgemm	*topk_rank*	*sgemm*	*topk_rank*	Total		
32	CPU	CPU	1.25s	0.06s	1.31s	-	41.12 ms/query
	GPU	CPU	0.08s	0.06s	0.14s	0.11s	7.91 ms/query
	GPU	GPU	0.08s	0.31s	0.39s	0.06 ms	12.14 ms/query
64	CPU	CPU	2.07s	0.12s	2.19s	-	34.20 ms/query
	GPU	CPU	0.15s	0.12s	0.27s	0.23s	7.85 ms/query
	GPU	GPU	0.15s	0.33s	0.48s	0.09 ms	7.51 ms/query
96	CPU	CPU	2.88s	0.18s	3.06s	-	31.84 ms/query
	GPU	CPU	0.23s	0.18s	0.41s	0.34s	7.83 ms/query
	GPU	GPU	0.23s	0.36s	0.59s	0.12 ms	6.13 ms/query

Finally, CUBA [6] proposes an architectural model where co-processors are encapsulated as function calls, as well as mechanisms to allow data physically residing on accelerator memory to be cached on CPU. CUBA assumes that the CPU has access to the co-processor memory mapped registers and to the co-processor local memory (which is not the case of the architecture we are considering). Moreover, the CUBA proposal discusses hardware changes (specifically, changes to the memory controller) whereas we operate at the runtime level.

3. MOTIVATIONAL EXAMPLE

In this section, we motivate the need for data-aware scheduling on heterogeneous platforms with a real application. The application we use is Supervised Semantic Indexing (SSI) classification [25].

SSI is an algorithm used to semantically search large document databases. It ranks the 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. Since documents and queries only contain a small fraction of possible words, each vector is sparse and has as many elements as the dictionary's size. Each vector element is the product of Term Frequency (TF) and Inverse Document Frequency (IDF) of the word that it corresponds to. TF is the number of times a word occurs in the document and IDF is the reciprocal of the number of documents that contain the word (thus IDF reduces the importance of commonly occurring words). Before classification can take place, the system must be trained. During this training process, a weight matrix is generated. By multiplying a query or document vector with the weight matrix, we obtain a smaller dense vector which contains relevant information for document-query classification. Each dense document and query vector is C elements long, where C is the number of concepts [25]. The classification process multiplies the query vector with all document vectors and identifies documents whose vectors produced the top k results.

The SSI classification process has two compute-intensive kernels which are good candidates for the many-core GPU. The first *(sgemm)* is the multiplication of the query vectors with all document vectors, essentially a dense matrix-matrix multiplication. With D documents in the database and Q simultaneous queries, the document matrix size is DxC and the query matrix size is QxC. The second kernel *(topk_rank)* must select, for each query vector, the top k best classification documents, that is, it selects the top k elements from the products of query vectors with document vectors. With millions of documents to search for each query, these two kernels take up 99% of the SSI execution time.

We motivate data-aware scheduling with three example runs of SSI classification. Our data set contains 1.6M documents and 128 conceptual categories. For each run, we vary the number of simultaneous queries performed (we consider 32, 64 and 96 queries). Each query requires the identification of 64 top classification documents from the document database. The document database contains documents selected from the Wikipedia [25]. For matrix multiplication, we use the Intel Math Kernel Library [26] on the CPU and the CUBLAS Library implementation of the *sgemm* function [27] on the GPU.

Table 1 shows the processing and data transfer times for three possible schedules of the two kernels used in SSI classification, as well as the overall throughput. The first schedule assumes both kernels are run on the CPU with no data transfer required. In the second schedule, the kernels are profiled and run on the computational element (either CPU or GPU) that has the smaller kernel processing time. In the third schedule, all kernels are run on GPU. However, data transfers are not performed before and after every kernel invocation, but only when required. In other words, the *topk_rank* kernel will be able to use the results of the previous call to *sgemm* without transferring them from the CPU.

As can be observed, dense matrix multiplication is much faster on GPU (by 12-15X), whereas *topk_rank* is slower on the GPU. However, as the number of queries increases, the speed of *topk_rank* on the GPU improves.

The poor performance of *sgemm* on the CPU affects the first schedule making it the worst for all the considered data sets. When the number of queries is small (32), the second schedule is preferable. As the number of queries increases, the third schedule tends to provide best performance. In particular, the throughputs achieved with the second and third schedules are comparable when 64 queries are processed in parallel. However, when the data set size increases to 96 queries, then the third schedule performs substantially better (the throughput achieved increases by 20%).

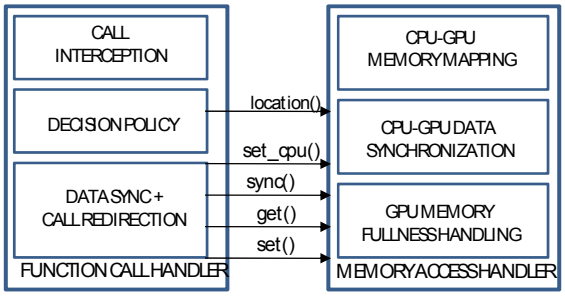

Figure 1: Block diagram of our runtime system.

Interestingly, what makes the third schedule preferable for large input sets is not the processing time, but the savings in terms of data transfer time. In fact, the *sgemm* call produces a matrix of size 1.6M×*Q* floats, which normally is transferred back to the CPU. If the runtime recognizes this and schedules the second kernel on the GPU even though the GPU is slower, the performance shown in the table is achieved. Note that the GPU-GPU schedule would never be preferable if data transfers were performed before and after each kernel invocation, such as would be the case if the runtime were handling legacy code, and was data-agnostic.

The key take-away is the following. First, the best schedule depends on data set size, determined at runtime for legacy kernels. Second, execution time alone is insufficient to achieve an optimal schedule. In particular, once a kernel executes on the GPU, the runtime should defer transferring back the data to the CPU until access to that data is performed. When subsequent kernels are invoked, the runtime should determine if it is worthwhile transferring back the data based on its predicted data transfer overhead, as well as the processing times of the kernel to run.

4. THE PROPOSED RUNTIME

In this section, we describe our runtime system design.

4.1 Overview of our Runtime

The primary goal of the runtime is to dynamically schedule computational kernels onto heterogeneous computing resources, namely the CPU and the GPU, in order to minimize the execution time of the overall application. To this end, the runtime aims to minimize kernel execution time as well as data transfer overheads. In effect, it hides the compute- and memory-heterogeneity from the programmer.

As mentioned above, the runtime operates at the granularity of a function call. The application runs by default on the CPU and may perform calls to well known kernels for which CPU or GPU implementations are provided. When one of these kernels is invoked, the runtime must determine the implementation to instantiate. This decision depends on two factors: kernel execution time and data transfer time. In turn, these factors depend on the size of the function call parameters and the location of the corresponding data. GPU kernel implementations assume that their parameters reside on the GPU memory: it is the responsibility of the runtime to hide this fact to the calling application, and to maintain a mapping between data structures residing on CPU and on GPU memories. As we will see, data is not transferred to the CPU memory at the end of each GPU kernel invocation, but only when required.

Note that each computational kernel – whether it targets the CPU or GPU – is essentially a "black box" to the runtime: *the only visible data transfers which can be optimized by the runtime pertain to the function arguments, and not to the data structures within the kernel itself*. In other words, the runtime aims at minimizing CPU-GPU data transfers; optimizing data transfers at different level of the GPU memory hierarchy is outside the scope of this work.

Figure 1 depicts our proposed runtime. It consists of two modules: *function call handler* and *memory access handler*. The function call handler intercepts kernel calls, determines which kernel implementations (CPU or GPU) to instantiate, and invokes them. The memory access handler maintains a mapping between CPU and GPU data structures, and handles data transfers and synchronizations. The services offered by the memory access handler are available to the function call handler through an API.

We now give more details on the two modules.

4.2 Function Call Handler

The function call handler intercepts predefined kernel calls and invokes proper library implementations depending on the call parameters and the data location. For each kernel `fn` having (read-only) input parameters `in_pars` and (write-only) output parameters `out_pars`, the module contains a function whose structure is exemplified in the pseudo-code below (`void` is used for illustration only).

The `mam` object (at lines 4, 7, 9, 11 and 13) represents the interface offered by the memory access module, that we will describe in more detail in the next section.

```
(1)   void fn(in_pars, *out_pars){
(2)     /* determine the best target for fn */
(3)     if(eval_loc(&fn,in_pars,out_pars)==CPU){
(4)       for (p in in_pars) mam->sync(p);
(5)       /* schedule on CPU */
(6)       cpu_fn(in_pars,out_pars);
(7)       for (p in out_pars) mam->set_cpu(p);
(8)     }else{
(9)       in_pars_d = out_pars_d = Ø;
(10)      for (p in in_pars)
(11)        in_pars_d U= mam->get(p,true);
(12)      for (p in out_pars)
(13)        out_pars_d U= mam->get(p,false);
(14)      /* schedule on GPU */
(15)      gpu_fn(in_pars_d, &out_pars_d);
(16)      for (p in out_pars) mam->set(p);
(17)    }
(18)  }
```

The `cpu_fn` and `gpu_fn` routines (at line 6 and 15, respectively) represent the CPU and GPU implementation of the intercepted kernel. Under GNU/Linux based operating systems, the function call handler can be dynamically linked to the application through the `LD_PRELOAD` directive. Pointers to `cpu_fn` and `gpu_fn` are obtained using the combination of `dlopen/dlsym` directives (the pointer to `cpu_fn` can also be obtained simply using `dlsym` and setting the handle to `RTLD_NEXT`).

The `eval_loc` routine (line 3) is also defined within the function call handler, and determines the best target for the intercepted function call. This decision is made by estimating the data transfer time of the input parameters and the kernel execution time on both CPU and GPU. We reiterate that the runtime

transfers (input) data only when they do not reside on the memory of the executing processor. `eval_loc` queries the memory access module for the location of each input parameter, and estimates the data transfer time based on the parameter size. In case of GPU execution, `eval_loc` considers the size and the location of the output parameters to determine whether the GPU has enough free memory to allocate them. In order to estimate the kernel execution time on both CPU and GPU, `eval_loc` uses profiling information. In particular, for all considered kernels, we measured the CPU and GPU execution time for different input parameters and we obtained the input size/execution time characteristic. At runtime, the `eval_loc` routine uses the actual input parameters to locate the operation point.

If the `eval_loc` routine establishes that the execution must happen on the CPU (lines 3-7), then the `cpu_fn` kernel must be invoked. Before its invocation, all input parameters must be synchronized (line 4). As we will see, `mam->sync` will have no effect if the CPU has an up-to-date copy of the data. After kernel execution, the output parameters are marked as residing on the CPU (line 7). This operation does not imply any data transfer.

If the kernel execution must take place on the GPU (lines 9-16), then `gpu_fn` is invoked (line 15). However, this kernel implementation operates on GPU memory. Therefore, a local copy of all input and output parameters (`in_pars_d` and `out_pars_d`) must be created (lines 9-13). For each parameter, the `mam->get` function returns the pointer to that copy (and, if necessary, allocates the corresponding memory on GPU). The last parameter of the `mam->get` call specifies whether the GPU must have an up-to-date copy of the data, which is necessary only for the input parameters. After kernel execution, the output parameters are marked as residing on the GPU (line 16). Again, this operation does not imply any data transfer.

4.3 Memory Access Handler

The goal of the memory access handler module is to orchestrate data transfers and synchronizations between CPU and GPU memory. In order to do so, it maintains a mapping between CPU and GPU memory regions. In particular, GPU global memory is seen as a set of *non overlapping data blocks,* each of them corresponding to a CPU data block. The mapping is stored in the *data block list*, a linked list of `data_block_t` structures, as represented below.

```
typedef enum {SYNCED,ON_CPU,ON_GPU} sync_t;

typedef struct {
    void *cpu_addr;
    void *gpu_addr;
    size_t size;
    sync_t sync;
    time_t timestamp;
}data_block_t;
```

Each data block has a CPU address `cpu_addr`, a GPU address `gpu_addr`, a `size` expressed in bytes, a synchronization status (`sync`) and a `timestamp` indicating the last access to the block. The synchronization status indicates whether the content of CPU and GPU blocks is synchronized (`SYNCED`) or whether the up-to-date copy of the data resides in CPU memory/GPU memory (`ON_CPU/ON_GPU`). Note that, since the application runs on the CPU and the runtime operates at the granularity of the function call, the memory access module allocates GPU memory (and updates the data block list) only when the runtime invokes the GPU implementation of an intercepted function.

The memory access handler offers primitives that are invoked by the runtime. The bulk of the CPU-GPU memory mapping's handling is performed within the `get` primitive, which is invoked by the runtime on all the parameters of a GPU kernel call.

```
void *get(void *cpu_addr, size_t size, bool update) throw Exception
```

Given a CPU memory block, `get` returns the pointer to the corresponding GPU memory block, and throws an exception if the block does not exist and cannot be allocated or transferred. If the parameter `update` is set to `true`, then the content of the GPU memory block must be up-to-date. This is typically valid when `get` is invoked on an input parameter of a function call, but is not required when this routine is called on an output parameter. For NVIDIA's GPUs, `get` uses `cudaMalloc` and `cudaMemcopy` [8] to perform memory allocations and data transfers.

When `get` is invoked, one of the following situations can occur (Figure 2). First, the required data block does not reside in GPU memory. In this case, a GPU memory allocation is performed, and a new entry is added to the data block list. The memory allocation is followed by a data transfer (from CPU to GPU) only if

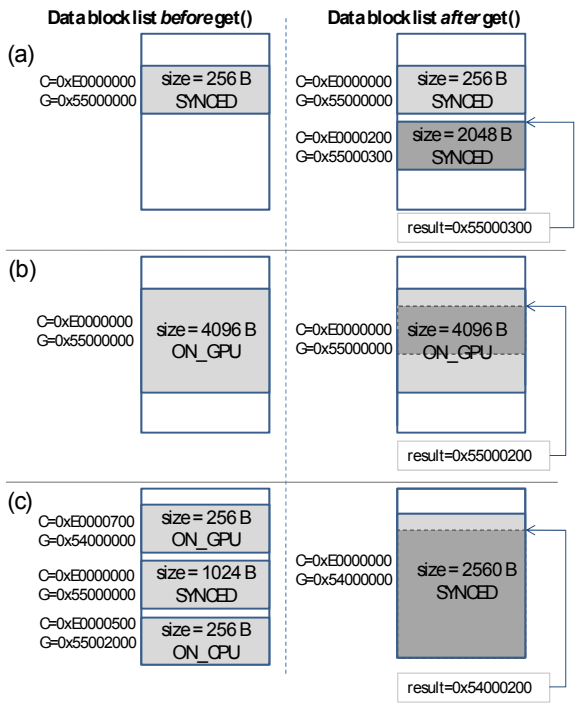

Figure 2: Examples of the outcome of invoking `get`(0xE0000200, 2048B, true) in different situations: (a) the requested data block is not yet allocated; (b) the requested data block is already present in GPU memory and its content is up-to-date; (c) the requested data block overlaps with several data blocks previously allocated. In all cases, we show the content of the data block list before (left hand side) and after (right hand side) the get's invocation, as well as the result of the operation (C=`cpu_addr`, G=`gpu_addr`) and the returned data block (highlighted in dark grey).

the `update` parameter of the `get` call is set to true. Second, the required data block already resides in GPU memory (possibly as part of a larger block). In this case, no memory allocation is required, and the content of the data block list is used to return the proper GPU address. A data transfer (from CPU to GPU) is performed only if the `update` parameter of the `get` call is set to true and the `sync` attribute of the block is equal to `ON_CPU`. In fact, no data transfer is needed if the GPU has already an up-to-date copy of the data. Finally, the requested data block – say B_{REQ} - spans multiple existing blocks B_i and possibly extends beyond them. In this case, it is necessary to allocate a new data block B_{NEW} which covers B_{REQ} and all the B_i. Each B_i can then be de-allocated and removed from the data block list. To understand why, consider that GPU kernels are a black box to the runtime, and that their parameters must point to contiguous memory regions. Again, the data transfer of block B_{NEW} from CPU to GPU is required only if the `update` parameter of the `get` call is set to true. However, if some B_i have attribute `sync` equal to `ON_GPU`, the portion of B_{NEW} overlapping them must be restored from GPU memory before their de-allocation. In Figure 2 (c) the following sequence of operations is assumed: first, block (`cpu_addr=0xE0000700`, `size=256`) is copied from GPU to CPU; second, all three blocks on the left hand side are de-allocated and removed from the data block list; finally, block (`cpu_addr=0xE0000000`, `size=2560`) is allocated and copied from CPU to GPU.

GPU kernel execution only affects GPU memory. The runtime does not enforce any GPU to CPU memory transfer after the invocation of a GPU kernel. Data consistency is ensured by invoking `set` on the output parameters of the GPU kernel call.

```
void set(void *cpu_addr) throw Exception
```

Given a CPU address, this routine sets the `sync` attribute of the corresponding data block to `ON_GPU`. An exception is thrown if such block cannot be found in the data block list.

When a kernel is invoked on CPU, the runtime must ensure that the CPU memory has an up-to-date copy of all input parameters. This is done with `sync`:

```
void sync(void *cpu_addr, size_t size) throw Exception
```

This function checks whether the data block list has one or more blocks B_i containing addresses in the range [`cpu_addr, cpu_addr+size`] and having attribute `sync` equal to `ON_GPU`. In this case, blocks B_i are copied to the CPU (and their attribute `sync` is set to `SYNCED`). Note that no action is required if the given address range is not mapped to GPU memory. An error during data transfer will cause an exception to be thrown.

After execution of a CPU kernel call, output parameters must be marked as residing on the CPU memory. This is accomplished by calling the `set_cpu` function.

```
void set_cpu(void *cpu_addr, size_t size)
```

This function sets the `sync` attribute of data blocks containing the given address range to `ON_CPU`. Again, no action is required if the data block list contains no such blocks.

As mentioned earlier, the `eval_loc` primitive in the function call handling module must obtain from the memory access module information about the location of the input parameters. This is achieved through the `location` function.

```
sync_t location(void *cpu_addr,size_t size)
```

`location` returns `ON_GPU` if the given address range belongs to a block B in the data block list, and the attribute `sync` of B is not equal to `ON_CPU`. In all other cases, `ON_CPU` is returned. Note that the goal of this function is to report whether invoking the `get` operation on the given address range would cause any GPU memory allocation and/or data transfer. This holds whenever `location` returns `ON_CPU`.

Finally, the memory access module provides a `free` primitive.

```
void free (void *cpu_addr, size_t size) throw Exception
```

`free` eliminates from the data block list all entries containing addresses from the given address range, and frees the corresponding GPU memory. This function is invoked in two circumstances: when the application de-allocates data, and when GPU memory runs full. In the latter case, the runtime uses the `timestamp` field in the `data_block_t` structure to determine the least recently used blocks. "Dirty" blocks are copied back to CPU before GPU de-allocation.

When running legacy applications, accesses performed *outside intercepted function calls* to address ranges mapped on GPU can originate data inconsistency problems. In the experiments presented in this work, we performed source code inspection and determined all accesses to variables which could potentially be modified by the intercepted function calls. We then modified the application by adding a call to `sync` before every memory read, and to `set_cpu` after every memory write to these variables. In [7] we describe operating system modifications to avoid this manual operation. The idea is to mark pages mapped to GPU as invalid, and to modify the page fault handler so that it will interact with our runtime and automatically call the proper function whenever a page fault is detected. In particular, handling will be performed within the runtime if the page fault involves a page mapped to GPU, whereas the page fault handler will resume its normal operation otherwise.

4.4 Additional Considerations

The runtime can be extended to support multiple GPUs or other devices connected to the CPU through the PCI-bus and having a local address space (e.g. FPGA-based accelerators). The extensions, which primarily involve the memory access module, depend on whether the design allows the same data to reside at the same time on multiple devices. The interested reader can find more discussion on this aspect in [7].

5. EXPERIMENTAL EVALUATION

In this section, we present some experimental results.

5.1 Methodology

We run our experiments on a heterogeneous workstation consisting of an Intel Xeon quad-core CPU and an NVIDIA Tesla C870 GPU. Table 2 shows the details of the architecture. As workloads, we used two real applications – K-means and SSI classification – as well as two synthetic applications consisting of various combinations of kernels, as summarized in Table 3.

The first application consists of two kernels, *Sort* (*quick sort* algorithm) and *Reduce*. *Sort* is implemented on the CPU using Intel TBB while *Reduce* is implemented using pthreads (in both cases, four threads are used). Both are implemented on the GPU

Table 2: Experimental setup.

	CPU	GPU
Model	Intel Xeon E5420	Tesla C870
Cores	4	128
Frequency	2.5 GHz	1.35 GHz
Memory size	12 GB	1.5 GB
Threading API	Pthreads, TBB	CUDA 2.3
Compiler	gcc -O3	nvcc 2.3 –O3

using CUDA 2.3. The GPU version of *Reduce* is from CUDA SDK [31].

K-means is the well-known clustering algorithm used in image segmentation [32]. We use Lloyd's algorithm [33] to select *k* means given *n* points (e.g., pixels in an image). Starting with an initial value for the *k* means, the algorithm proceeds iteratively. Each iteration consists of three parallelizable kernels that we call *K1, K2* and *K3*. *K1* calculates the Euclidean distance between the *n* points and the current *k* means. *K2* picks the closest mean for each point, and *K3* updates the values of the *k* means by averaging all points closest to each mean. Since *K3* could only be parallelized into *k* threads, and *k* is small (under 64), it was always faster on the CPU. We implemented *K1* and *K2* on both the CPU and GPU using Intel's MKL [26] and CUDA 2.3 respectively.

The third application consists of two kernels, *SpMV* and *topk_rank*. *SpMV* [31] performs sparse matrix-vector multiplication. For *topk_rank*, the same kernel used in the example of Section 3, we use our own implementation on both CPU (using pthreads) and GPU (using CUDA).

Finally, SSI classification uses two kernels (dense matrix multiplication and topk_rank) and has been described in Section 3.

Table 3: Benchmarks.

Apps	Description	Input Size
Sort + Reduce	Synthetic benchmark with parallel sorting and parallel reduction kernels	Data size from 4K elements to 1024K elements
K-means	Clustering algorithm used in image segmentation	1K to 1M pixels clustered into 32 regions
SpMV+ topk-rank	Synthetic benchmark with sparse matrix-dense vector multiplication and top *k* ranking	Sparse matrices with 100-700K rows/columns, up to 3.9M non-zeros
SSI	Supervised Semantic Indexing of documents based on text queries	1.8M documents with 32-96 simultaneous queries

For all applications, we measure wall-clock processing as well as data transfer times. In the experiments that use our data-aware runtime system, we also accounted for the overhead due to call interception and runtime scheduling.

5.2 Results

In this section, we report our findings using our data-aware runtime for the above applications.

5.2.1 Sort and Reduce

Figure 3 shows the performance of running Sort and Reduce on CPU and GPU separately. The GPU performance bars show the split costs for real processing, memory allocation and data transfer. We see that Sort on GPU is slightly faster for small data

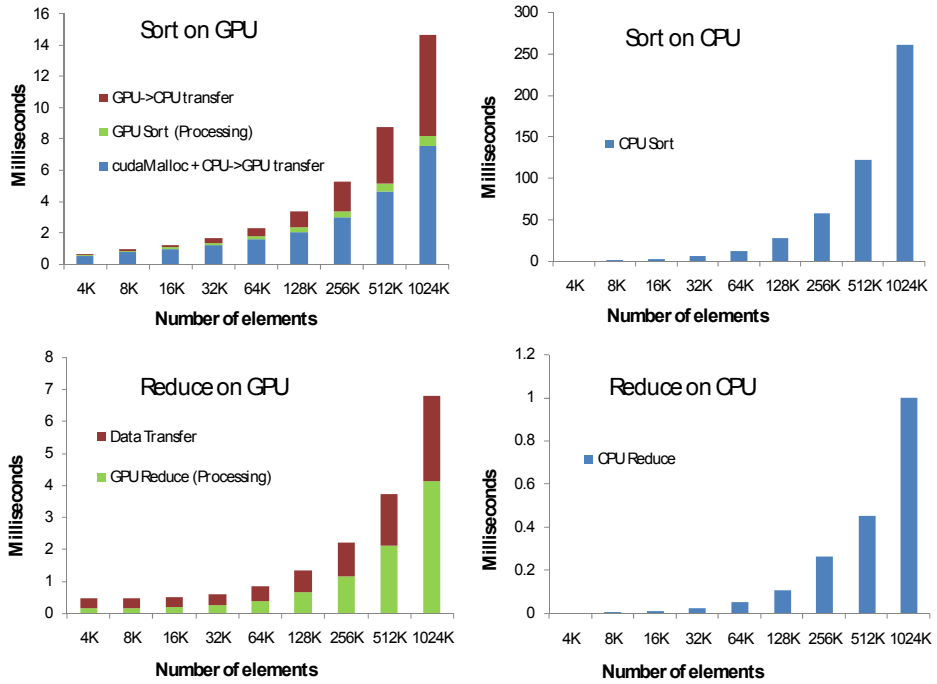

Figure 3: Processing and data transfer time for *Sort* and *Reduce* on GPU (left) and CPU (right). The GPU is faster for *Sort* while the CPU is faster for *Reduce*.

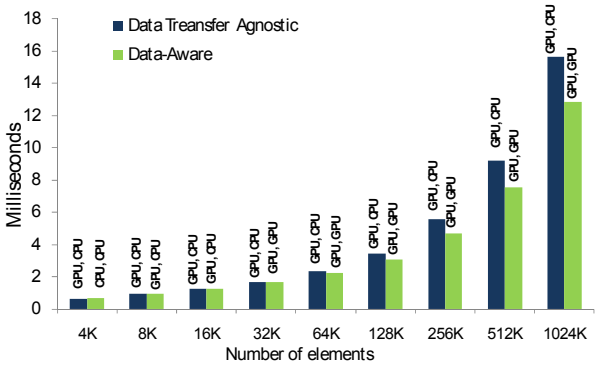

Figure 4: Execution time for *Sort* and *Reduce*. The labels on the bars indicate where the kernels were scheduled.

sizes (4K elements) and much faster as the data size increases. For Reduce, the CPU multithreaded version is faster than the GPU implementation.

When running an application consisting of multiple kernels, a data-agnostic scheduler assigns each kernel to the computational unit that offers the best performance, in this case the GPU for Sort and the CPU for Reduce. However, being unaware of data location, such a scheduler leads to data transfers before and after each GPU kernel invocation. The data-agnostic runtime must do this since it is unaware of which kernel may be invoked next, as is the case for legacy code. As can be observed, the transfer time is not trivial for large data sizes. Our runtime keeps track of data location, delays data transfers and takes the cost of data transfers into consideration when performing online scheduling decisions.

Figure 4 compares the performance of a data agnostic runtime with our data-aware runtime, where the data-agnostic runtime schedules kernels on the processor that is estimated to be faster, regardless of data location. In this case, a data-agnostic scheduler would always pick the GPU for Sort and CPU for Reduce. Our data-aware runtime schedules both kernels on the CPU when data size is small (4K elements), but picks the GPU for Sort and CPU for Reduce for intermediate data sizes (8K-16K), and runs both kernels on GPU for larger data. While there is a small performance loss due to our runtime overhead (under 2%) for small data sizes, we achieve around 20% performance improvement when these kernels work with 256K or more elements.

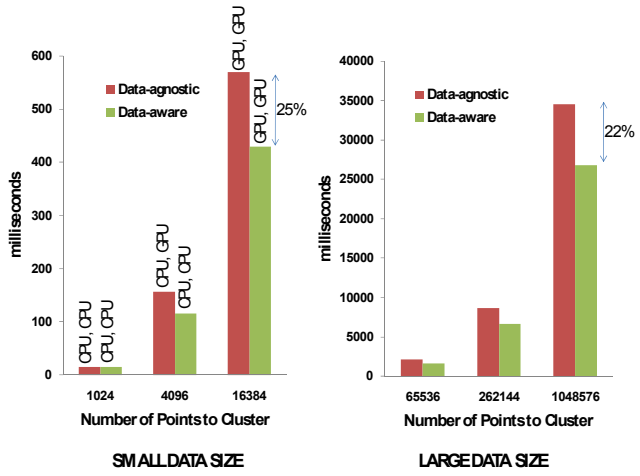

Figure 5: Data-agnostic and Data-aware scheduling for K-means with small (left) and large data sizes (right). Data-aware scheduling makes the application 25% faster than a data-agnostic runtime.

5.2.2 *K-means*

We recall that K-means has two candidate kernels *K1* and *K2*. The third kernel *K3* is always faster on the CPU, and with negligible data transfer into and out of *K3*, it is always scheduled on the CPU. We segmented random images of sizes ranging from 1K pixels to 1M pixels into 32 clusters (i.e., $k = 32$). We found that for small images (specifically 1K and 4K pixels), the CPU was faster than the GPU for kernel *K1* (it used MKL sgemm for most of its Euclidean distance computation), but the GPU was faster (with its CUBLAS sgemm implementation [27]) for larger images. Our custom implementation of Kernel *K2* was faster on the GPU for images 4K or larger. Figure 5 shows the performance of K-means with data-agnostic and data-aware runtimes for small images and large images. Labels above the bars indicate the schedule for the two kernels. While the two runtimes schedule the kernels the same way for 16K and larger images, the performance improvement with the data-aware runtime is due to the optimization of data transfers. Specifically, after kernel *K1* runs on the GPU, the runtime postpones the data transfer back to the CPU until *K2* has been scheduled. From the figure, we see the data-aware runtime improves performance by up to 25% for both large and small data sets. Figure 6 shows the performance profile of the 3 different kernels in K-means. The data-aware runtime profile is shown on the left, and the data-agnostic on the right. We see that the data transfer portion of the profile is significantly reduced by the data-aware runtime resulting in the 25% performance improvement.

5.2.3 *SpMV and topk_rank*

Sparse matrix (*SpMV*) performance on the CPU and GPU depends on the number of non-zeros in the matrix. For our experiments,

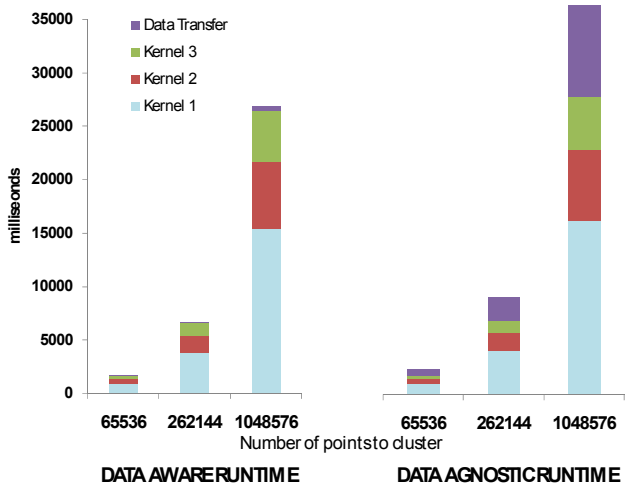

Figure 6: Performance profile of K-means, and data transfer time for data-agnostic and data-ware runtimes.

Table 4: Results for synthetic application with SpMV and topk_rank kernels.

Matrix	Rows/Cols	Non-zeros	# Vectors	Data-Agnostic Runtime		Data-aware Runtime	
				Schedule	Time (ms)	Schedule	Time (ms)
SparseM1	100	460	64	CPU, CPU	1.02	CPU, CPU	1.02
			128	CPU, GPU	1.75	CPU, GPU	1.75
SparseM2	36057	341088	64	GPU, GPU	94.51	GPU, GPU	74.46
			128	GPU, GPU	186.25	GPU, GPU	147.29
SparseM3	682862	3871773	64	CPU, GPU	1540.15	CPU, GPU	1540.15
			128	CPU, GPU	3091.81	CPU, GPU	3091.81

we use three matrices (ranging in size from 100 to nearly 700K rows/columns, and up to 3.9M non-zeros, obtained from [36]) each multiplied by 64 and 128 dense vectors, as shown in Table 4. For each case, we multiply a given sparse matrix with the vectors, and select the top 64 elements of each result vector. The data-agnostic runtime selects a schedule solely based on the estimated performance of the kernel, while our data-aware runtime selects the schedule based on estimated performance as well as the estimated data transfer overhead. For this benchmark, although our runtime chooses the same schedule as the data-agnostic runtime, it has better performance (for SparseM2) due to the fact that it defers data transfer and figures out it can avoid them. For SparseM2, once the SpMV has executed on the GPU, the data is not transferred back to the CPU until the next kernel is encountered a decision made regarding its schedule. We see improvements of up to 21% for SparseM2, but do not significantly affect the performance of the other matrices (our runtime overhead is under 2%).

5.2.4 Supervised Semantic Indexing (SSI)

We ran SSI with 32, 64 and 96 parallel queries, semantically searching the Wikipedia database consisting of 1.8M documents. SSI has two compute-intensive kernels: matrix multiplication *sgemm* and *topk_rank*. Table 5 shows the schedules and overall SSI performance in milliseconds per query for each case under a data-agnostic and our data-aware runtime. We see a performance improvement of 21.7% for 96 queries, of 4.4% for 64 queries and a negligible degradation for the small data set.

6. CONCLUSION, FUTURE DIRECTIONS

We presented a runtime for heterogeneous platforms consisting of one or more multi-core CPUs coupled with one or more many-core GPUs via a non-coherent interconnect. The CPU and GPU do not have shared memory. The runtime provides a unified memory view to the programmer, and aims at enabling legacy programs to run seamlessly on the heterogeneous platform with higher performance.

The key contribution is making the runtime data-aware. The proposed runtime schedules computations as well as data transfers taking into account the estimated performance and the time required to move data. In doing so, it may schedule a kernel on the slower processor simply because of data proximity. It also defers transferring data until necessary; thus, a kernel that runs on the GPU will not have its data transferred back to the CPU even though the runtime is unaware of when the data will be used in future. Rather, when another kernel requires those data, the runtime decides if they should be moved to a different processor, or the kernel should be scheduled on the processor hosting the data.

We implemented the data-aware runtime and evaluated it on a heterogeneous platform with a quad-core x86 CPU and an NVIDIA Tesla C870 (128-core) GPU. For synthetic as well as real applications, our runtime shows a performance improvement of up to 25% when compared to a runtime that schedules in a data-agnostic manner.

REFERENCES

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

[2] T. J. Knight *et al*, "Compilation for Explicitly Managed Memory Hierarchies," in Proc. of PPoPP 2007, San Jose, CA.G.

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

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

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

[6] I. Gelado *et al*, "CUBA: An Architecture for Efficient CPU/Co-processor Data Communication," in Proc. of ICS'08, Island of Kos, Greece.

[7] M. Becchi, S. Cadambi and S. T. Chakradhar, "Enabling Legacy Applications on Heterogeneous Platforms," in Proc. of HotPar 2010, Berkeley, CA, June 2010.

[8] CUDA documentation: http://www.nvidia.com/object/cuda_develop.html.

Table 5: Results for SSI with data-aware scheduling for 1.8M document database.

# Parallel Queries	Data-Agnostic Runtime		Data-aware Runtime	
	Schedule	Performance	Schedule	Performance
32	GPU, CPU	7.91 ms/query	GPU, CPU	7.91 ms/query
64	GPU, CPU	7.85 ms/query	GPU, GPU	7.51 ms/query
96	GPU, CPU	7.83 ms/query	GPU, GPU	6.13 ms/query

[9] http://www.supermicro.com/products/nfo/files/GPU/GPU_White_Paper.pdf: "Shattering the 1U Server Performance Record".

[10] AMD, AMD Stream SDK User Guide v 2.0, 2009.

[11] Intel, Intel Threading Building Blocks 2.2: http://www.threadingbuildingblocks.org.

[12] A. Ghuloum *et al*, "Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture", Intel Technology Journal 11, 4, 333-348, Nov 2007.

[13] D. Tarditi, S. Puri and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in Proc. of the 2006 ASPLOS, October 2006.

[14] Intel RapidMind, http://software.intel.com/en-us/articles/rapidmind.

[15] Peakstream, "Peakstream Stream Platform API C++ Programming Guide v 1.0", May 2007.

[16] PGI, PGI Accelerator Compilers, http://www.pgroup.com/resources/accel.htm.

[17] CAPS, HMPP Workbench, http://www.caps-entreprise.com/hmpp.html.

[18] HPC Project, Par4All, http://www.par4all.org.

[19] J. A. Stratton, S. S. Stone and W-m. W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels from Multi-Core CPUs," in Proc. of the 2008 Workshop on Languages and Compilers for Parallel Computing, 2008.

[20] G. Diamos *et al*, "GPUocelot – A binary Translator Framework for GPGPU" http://code.google.com/p/gpuocelot.

[21] S.-W. Liao *et al*, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in Proc. of the 4th Conference on CGO, March 2006.

[22] A. Munshi, "OpenCL Parallel Computing on the GPU and CPU", in ACM SIGGRAPH 2008.

[23] K. O'Brien *et al*, "Supporting OpenMP on Cell," in International Journal on Parallel Programming, 36, 289—311, 2008.

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

[25] 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.

[26] Intel MKL: http://software.intel.com/en-us/intel-mkl.

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

[28] T. Kosar, "A New Paradigm in Data Intensive Computing: Stork and the Data-Aware Schedulers," in Proc. of Challenges of Large Applications in Distributed Environments, 2006.

[29] J. Bent *et al*, "Coordination of Data Movement with Computation Scheduling on a Cluster," in Proceedings of Challenges of Large Applications in Distributed Environments, 2005.

[30] G. Khanna, "A Data-Locality Aware Mapping and Scheduling Framework for Data-Intensive Computing", MS Thesis, Dept. of Computer Science and Engineering, The Ohio State University, 2008.

[31] Nvidia, "CUDA SDK Code examples", http://www.nvidia.com/object/cuda_get.html.

[32] J. B. MacQueen, "Some methods for classification and analysis of multivariate observation," in Proc. of the Berkeley Symposium on Math. Stat. and Prob., pp 281–297.

[33] S.P. Lloyd, "Least squares quantization in PCM," IEEE Transactions on Information Theory 28 (2): pp 129–137.

[34] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multicore architectures," in Proc. of HPPC'08, Las Palmas de Gran Canaria, Spain, August 2008.

[35] C. Augonnet *et al*, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in Proc. of the 15th International Euro-Par Conference, Delft, The Netherlands, August 2009.

[36] http://www.cise.ufl.edu/research/sparse/matrices/