An Extension of the StarSs Programming Model for Platforms with Multiple GPUs

Eduard Ayguadé[1], Rosa M. Badia[1,2], Francisco D. Igual[3], Jesús Labarta[1], Rafael Mayo[3], and Enrique S. Quintana-Ortí[3]

[1] Barcelona Supercomputing Center – Centro Nacional de Supercomputación (BSC–CNS) and Universitat Politècnica de Catalunya, Nexus II Building, C. Jordi Girona 29, 08034–Barcelona, Spain
{eduard.ayguade,rosa.m.badia,jesus.labarta}@bsc.es
[2] Consejo Superior de Investigaciones Científicas (CSIC), Spain
[3] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain
{figual,mayo,quintana}@icc.uji.es

Abstract. While general-purpose homogeneous multi-core architectures are becoming ubiquitous, there are clear indications that, for a number of important applications, a better performance/power ratio can be attained using specialized hardware accelerators. These accelerators require specific SDK or programming languages which are not always easy to program. Thus, the impact of the new programming paradigms on the programmer's productivity will determine their success in the high-performance computing arena. In this paper we present GPU Superscalar (GPUSs), an extension of the Star Superscalar programming model that targets the parallelization of applications on platforms consisting of a general-purpose processor connected with multiple graphics processors. GPUSs deals with architecture heterogeneity and separate memory address spaces, while preserving simplicity and portability. Preliminary experimental results for a well-known operation in numerical linear algebra illustrate the correct adaptation of the runtime to a multi-GPU system, attaining notable performance results.

Keywords: Task-level parallelism, graphics processors, heterogeneous systems, programming models.

1 Introduction

In response to the combined hurdles of maximum power dissipation, large memory latencies, and little instruction-level parallelism left to be exploited, all major chip manufacturers have finally adopted multi-core designs as the only means to exploit the increasing number of transistors dictated by Amdahl's Law. Thus, desktop systems equipped with general-purpose four-core processors and graphics processors (GPUs) with hundreds of fine-grained cores are routine today [8]. On the other hand, NVIDIA and AMD GPUs, and other accelerators like the IBM Cell B.E. or ClearSpeed boards, have demonstrated that a high performance/power ratio can be attained for certain applications using their specialized

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 851–862, 2009.

hardware. Thus, it is natural to think that the amount of transistors available in future systems will make feasible to integrate in the chip functionalities similar to a hardware accelerator, which so far were external to the processor.

While novel and interesting heterogeneous architectures are expected for the future, we believe that it is the software (i.e., how easy is to program the new parallel architectures) that will determine their success or failure. The StarSs programming model addresses the programmability problem exploiting task-level parallelism [4,12,10]. It consists of a few OpenMP-like pragmas, a source-to-source translator, and runtime system that schedules tasks to execution preserving dependencies among tasks. Instantiations of the StarSs programming model include GRIDSs (for the Grid), CellSs (for the Cell B.E.), and SMPSs (for multi-core processors). In this paper we extend StarSs with a new instantiation, GPUSs (GPU Superscalar), with the following specific contributions:

– *Heterogeneity:* The target architecture for GPUSs is fundamentally different: a heterogeneous system with a general-purpose processor and multiple GPUs. Although a similar approach has been investigated by some of the authors as part of the FLAME project [13], the focus there was in the specific domain of dense linear algebra. GPUSs is a general-purpose programming model, also valid for non-numerical applications.
– *Separate memory spaces:* We investigate the use of techniques such as software caches, cache coherence mechanisms, and scheduling of bundles of tasks from CellSs and FLAME. The purpose is to hide the existence of multiple memory address spaces (as many as GPUs plus that of the general-purpose processor) from the programmer while still delivering high performance. The internal memory hierarchy of the graphics processor should be explicitly managed by the programmer of the kernels in order to attain high performance. This management is out of the scope of our runtime and programming model.
– *Simplicity:* GPUSs inherits the simplicity of StarSs, consisting of a reduced number of OpenMP-like pragmas, and the corresponding tailored versions of the StarSs translator and runtime.
– *Portability:* Although we illustrate a GPU-based implementation, most of the techniques shown can be easily applied to other type of multi-accelerator platform without major modifications of the runtime or the user codes.

Few works target the automatic parallelization of codes focusing platforms with multiple hardware accelerators. In [13] the authors propose an extension of the SuperMatrix runtime to execute linear algebra codes on systems with multiple graphics processors. Although targeting systems with one GPU, [7] presents a compiler framework for automatic source-to-source translation of OpenMP applications into optimized GPU code.

The rest of the paper is organized as follows. In Section 2 we introduce the Cholesky factorization as a motivating example. Practical aspects of an algorithm to compute this operation are also given in that section. Sections 3 and 4 contain the major contributions of this work. In the first one, we illustrate the use of the GPUSs *constructs* to parallelize a blocked algorithm for the Cholesky factorization; in the second, we describe the main features of the GPUSs runtime which put these ideas into practice. Section 5 contains initial results for this operation on a desktop system with two Intel QuadCore processors connected to a TESLA s1070 (four NVIDIA GT200 processors). Finally, concluding remarks and future work are summarized in Section 6.

2 Computing the Cholesky Factorization

To illustrate the GPUSs extension, we will use an important operation for the solution of linear systems of equations: the Cholesky factorization. This is the first step in solving the system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a symmetric positive definite (SPD) matrix. While numerical linear algebra operations like the Cholesky factorization are appealing because of their use in practical applications, we note that GPUSs targets the parallelization of general numerical and non-numerical codes. Here we chose this particular operation because it exhibits tasks parallelism and an elaborated pattern of data dependencies among tasks.

The Cholesky factorization of a dense SPD matrix $A \in \mathbb{R}^{n \times n}$ is defined as

$$A = LL^T, \tag{1}$$

```
#define NT ...
#define TS ...

void Cholesky( float * A ) {
  int i,j,k;

  for (k = 0; k < NT; k++) {
    chol_spotrf( A[k*NT+k] );                 // Factorize diagonal block

    for (i = k+1; i < NT; i++)
      chol_strsm( A[k*NT+k], A[k*NT+i] );     // Triangular solves

    for (i = k+1; i < NT; i++) {              // Update trailing submatrix
      for (j = k+1; j < i; j++)
        chol_sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i] );
      chol_ssyrk( A[k*NT+i], A[i*NT+i] );
    }
  }
}

int main( void ) {
  int i, j, k;

  float *A[NT][NT];                           // Allocate matrix of NT x NT blocks,
                                              // of dimension TS x TS each
  for (i = 0; i < NT; i++)
    for (j = 0; j <= i; j++)
      A[i][j] = ( float * ) malloc( TS*TS*sizeof( float ) );

  Init_matrix( A );                           // Initialize elements of the matrix

  Cholesky( A );                              // Compute the Cholesky factor
}
```

Fig. 1. Blocked algorithm for computing the Cholesky factorization

```
// LAPACK spotrf wrapper
void chol_spotrf( float *A )
{
  int info = 0;
  int ts   = TS;
  spotrf( "Lower", &ts, A, &ts,
          &info );
}
```

```
// BLAS trsm wrapper
void chol_strsm( float *T,
                 float *B )
{
  float sone = 1.0;
  int ts     = TS;
  strsm( "Right", "Lower",
         "Transpose", "Non-Unit",
         &ts, &ts,
         &sone, T, &ts,
                B, &ts );
}
```

```
// BLAS gemm wrapper
void chol_sgemm( float *A,
                 float *B,
                 float *C )
{
  float sone       =  1.0,
        minus_sone = -1.0;
  int ts           =  TS;
  sgemm( "No Transpose", "Transpose",
         &ts, &ts, &ts,
         &minus_sone, A, &ts,
                      B, &ts,
         &sone,       C, &ts );
}
```

```
// BLAS syrk wrapper
void chol_ssyrk( float *A,
                 float *C )
{
  float sone        =  1.0,
        sminus_one  = -1.0;
  int ts            = TS;
  ssyrk( "Lower", "No Tranpose",
         &ts, &ts,
         &sminus_one, A, &ts,
         &sone,       C, &ts );
}
```

Fig. 2. Building blocks for the blocked algorithm for computing the Cholesky factorization

where $L \in \mathbb{R}^{n \times n}$ is a lower triangular matrix known as the Cholesky factor of A. (A can be decomposed as $A = U^T U$, with $U \in \mathbb{R}^{n \times n}$ being upper triangular.)

A blocked algorithm for the Cholesky factorization is given in routine Cholesky in Figure 1 together with the main function that allocates the matrix and invokes this routine. Following the common practice, the factorization algorithm overwrites the lower triangular part of the array A with the entries of L, while the strictly upper triangular part of the matrix remains unmodified. (Note here that both in the driver and the factorization routine, the matrix is considered to be stored by blocks; i.e., entries in the same block are stored in contiguous positions in memory. This is known to yield better performance for numerical applications; see, e.g., [9,5].) Figure 2 shows that `chol_spotrf`, `chol_strsm`, `chol_sgemm`, and `chol_ssyrk` are simple wrappers to routines `spotrf`, `strsm`, `sgemm`, and `ssyrk` from LAPACK (the first one) and BLAS (the last three ones) [1,6]. Highly tuned implementations of the BLAS routines are provided by most hardware vendors; examples of interest to this work include Intel MKL and NVIDIA `CUBLAS`.

3 The GPUSs Framework

3.1 Target Platform

Multi-accelerator systems are the natural target platform for the GPUSs framework. A generic multi-accelerator system consists of a general-purpose processor (possibly with several cores) called *host*, connected to a number of hardware accelerators, or *devices*. We assume that each device can only access its own local memory space. In our case, the devices are programmable GPUs which communicate with the CPU via a PCIExpress bus. Direct communication between devices is not possible, so that data transfer between them must be performed through the host memory (main memory).

Most modern hardware accelerators are designed as *many-core* systems, replicating specialized fine-grained cores, and featuring an internal memory hierarchy. It is important to point out that exploiting the hardware parallelism due to the existence of multiple fine-grained cores inside the accelerator is out of the scope for GPUSs. Our approach considers each accelerator as a single execution unit (or coarse-grained core), capable of efficiently executing pieces of code (or *kernels*) written by the programmer. Thus, GPUSs is not aware of the internal architecture of the hardware accelerator. It only exploits the parallelism derived from the existence of multiple hardware accelerators connected to the system.

GPUSs can be used to parallelize a code consisting of several invocations to CUDA kernels initially designed to be executed on a single GPU, adapting it to a multi-GPU system. Tuning techniques applied inside those kernels (use of shared memory, coalesced accesses to target memory, absence of bank conflicts in the access to shared memory,. . .) are CUDA-specific improvements which will affect the global performance of the parallel execution. However, those details are transparent to GPUSs, whose goal is to efficiently dispatch the execution of kernels to different accelerators, reducing both the number of data transfers and device idle times.

3.2 Programming Model

The programming model introduced by StarSs and extended by GPUSs allows the automatic parallelization of sequential applications. A runtime system is in charge of using the different hardware resources of the platform (the multi-core general-purpose processor and the GPUs) in parallel to execute the annotated sequential code.

It is responsibility of the programmer to annotate the sequential code to indicate that a given piece of code will be executed on a GPU. These annotations themselves do not indicate a parallel region, but a function which can be run on an accelerator. In a system with multiple accelerators, the runtime extracts the parallelism by building a data dependency graph (in which each node represents an instance of an annotated function, also called a *task*, and edges denote data dependencies between tasks) and by executing independent tasks on the different accelerators in parallel.

GPUSs basically provides two OpenMP-like constructs to annotate code. The first one, directly inherited from StarSs, is used to identify a unit of work, or task, and can be applied to tasks that are just composed of a function call, as well as to headers or definitions of functions that are always executed as tasks:

```
#pragma css task [clause-list]
{function-header | function-definition | function-call}
```

When the program calls a function annotated in this way, the runtime will create an explicit task. This construct is complemented with clauses `input`, `output`, and `inout`, which identify the directionality (input, output, or both, respectively) of the arguments to the function. The clauses are used by the GPUSs runtime to track dependencies among tasks and decide at run time whether a task can be scheduled/issued for execution.

In our particular example, we can use this construct to annotate the LAPACK and BLAS wrappers as follows:

```
#pragma css task inout ( A[TS][TS] )
void chol_spotrf( float *A );

#pragma css task input ( T[TS][TS] ) inout ( B[TS][TS] )
void chol_strsm( float *T, float *B );

#pragma css task input ( A[TS][TS] ) inout ( C[TS][TS] )
void chol_ssyrk( float *A, float *C );

#pragma css task input ( A[TS][TS], B[TS][TS] ) inout ( C[TS][TS] )
void chol_sgemm( float *A, float *B, float *C );
```

Although we have annotated the header declaration of the wrappers here, the effect is the same that would be obtained by annotating the function call or its definition. We also indicate the dimension of the non-scalar arguments in clauses; this is trivial in our example, as all matrix blocks are of size `TS` × `TS`. Here `TS` could also be an argument to the function, with the appropriate directionality indication.

The second construct follows a recent proposal to extend the OpenMP tasking model for heterogeneous architectures in [2], and has been incorporated in GPUSs. The construct adopts the form

```
#pragma css target device( device-name ) [clause-list]
```

The `target` construct specifies that the execution of the task should be offloaded to the device specified by `device-name` (and as such, its code must be handled by the appropriate compiler back-end and its execution appropriately managed by the runtime). Tasks which are not annotated with this construct are executed on the host. For Nvidia GPUs, the device name is `cuda`. Some additional clauses can be used with this pragma, to specify data movement between memory spaces of the shared variables inside the task. In particular

```
copy_in( data-reference-list )
copy_out( data-reference-list )
```

will move the variables in `data-reference-list` from host to device memory or vice-versa, respectively. In other words, when the corresponding task is ready to be issued, the runtime will transfer the variables in the `copy_in` list from host memory to device memory. Once the execution of the tasks is completed, variables in the `copy_out` list will be retrieved from device to host memory.

Applying the previous construct to the Cholesky factorization is simple:

```
#pragma css task inout ( A[TS][TS] )
void chol_spotrf( float *A );

#pragma css target device( cuda ) copy_in( T[TS][TS], B[TS][TS] )\
                                  copy_out( B[TS][TS] )
#pragma css task input ( T[TS][TS] ) inout ( B[TS][TS] )
void chol_strsm( float *T, float *B );

#pragma css target device( cuda ) copy_in( A[TS][TS], C[TS][TS] )\
                                  copy_out( C[TS][TS] )
#pragma css task input ( A[TS][TS] ) inout ( C[TS][TS] )
void chol_ssyrk( float *A, float *C );

#pragma css target device( cuda ) copy_in( A[TS][TS], B[TS][TS],\
                                           C[TS][TS] )\
                                  copy_out( C[TS][TS] )
#pragma css task input ( A[TS][TS], B[TS][TS] ) inout ( C[TS][TS] )
void chol_sgemm( float *A, float *B, float *C );
```

In this case, we have decided to offload the execution of the BLAS wrappers `chol_strsm`, `chol_ssyrk`, and `chol_sgemm` to the GPUs. The LAPACK wrapper `chol_spotrf` will be executed on the host. The examples above use optimized implementations of these BLAS kernels for the GPU (e.g., in the Nvidia implementation `CUBLAS`). Thus, we accommodate the possibility of executing a given task in the host, which may be more efficient for certain tasks.

Function definitions are not restricted to wrappers to library routine calls, as in the examples above. In GPUSs, native CUDA code could also be included in the body of the annotated functions. In this case, the programming model allows the developer to explicitly define some CUDA-specific execution parameters (basically, grid and block sizes; see [8].)

The bottom-line is that the sequential code in Figure 1 does not need to be modified to execute it in parallel on a multi-GPU platform. With the previous annotations, GPUSs automatically parallelizes the execution of the factorization, issuing tasks (or bundles of them) to the available execution units.

4 The GPUSs Runtime

4.1 Adapting the CellSs Runtime to the Tesla System

Although being very different from the architectural viewpoint , there are many conceptual similarities between the Cell and the Nvidia Tesla architectures, which allow the adaptation of the runtime developed for the Cell to a system with multiple hardware accelerators. In general, many of these similarities also hold for other multi-accelerator systems.

Both architectures are heterogeneous: the Cell consists of a general-purpose processor, or PPE, that orchestrates execution on a set of specialized fast cores, or SPEs. All processors can communicate through a fast interconnection bus (EIB), using small memory pools for each SPE, and a global memory space accessible to all processors. On the other hand, the multi-GPU system consists of a central general-purpose processor with a memory pool (RAM) and a number of GPUs (NVIDIA TESLA) connected to the host through a PCIExpress bus.

Some of the practical differences between both architectures have a direct implication in the design and adaptation of the runtime. First, the Cell PPE is much slower than the SPEs. Thus, the workload assigned to the PPE must be carefully designed to avoid penalizing the global performance of the runtime. In particular, critical decisions such as scheduling or resource assignation must be simplified, adopting trade-offs between simplicity and performance.

In addition, the local memory spaces for each GPU are considerably larger than the corresponding local stores of each SPE, and GPUs usually attain higher performance for large data streams [3]. This dictates the use of a coarser task granularity in GPUSs, which gives more time to the CPU to apply more elaborated scheduling techniques that may boost performance.

The third main difference lies in the interconnect between host and devices. Here, the peak bandwidth of the Cell EIB (25 GB/s per channel) is much higher than that of the PCIExpress bus (0.5 GB/s per lane in the Gen2 specification.) Moreover, direct communication between SPEs can be performed in the Cell, but this technique cannot be applied for the TESLA system where all transfers between GPUs must be done through the main memory. The penalty introduced by the use of the PCIExpress bus urges a reduction of data transfers. The impact of data transfers in multi-GPU systems has already been analyzed [13], and similar techniques can also be applied to the GPUSs runtime implementation.

4.2 The GPUSs Runtime

Despite the architectural differences between the Cell B.E. and a SMP processor, the two runtime systems developed for those architectures in the StarSs framework (CellSs and SMPSs) share many features that have been inherited by the GPUSs implementation described.

Both runtime systems are divided in two main modules. The first one is devoted to the execution of the annotated user code, task identification and generation, data renaming and scheduling. The second module manages the data movements between memory spaces (only in the CellSs implementation) and task execution. The actual implementation of those modules varies for each system; more details can be found in [12] and [4]. We will focus on the main differences and particularities introduced in the GPUSs runtime.

Similarly to the CellSs runtime (see [4,11]), three *actors* play a fundamental role in the execution of an application using the GPUSs runtime:

– A *master* thread executes the user code, intercepting calls to annotated functions, generating *tasks*, and inserting them in a Task Dependency Graph.
– A *helper* thread consumes tasks as the GPUs become idle, mapping them to the most suitable device taking into consideration data locality policies.
– A set of *worker* threads (one per GPU) which wait for available tasks, perform the necessary data transfers between RAM memory and the corresponding GPU, invoke the task call on GPU, and retrieve the results (if necessary). There is a key difference in the worker infrastructure between the Cell and

the NVIDIA TESLA system: GPUs are passive processing elements, waiting for tasks ready for execution, but without any other processing capability; thus, worker threads are executed on the CPU, while in the Cell B.E. worker threads run in the accelerator/SPEs. Therefore, the GPUSs runtime is executed entirely on the host, not divided into host and device as in the CellSs system.

As in the CellSs runtime, the main program communicates with the corresponding *worker* thread by using event signaling when a GPU becomes idle, and a new task is ready for execution; the *worker* thread receives the necessary information to identify and invoke each ready task, and locate the necessary parameters for the execution of the task. Once the task has been executed by the proper GPU and data has been transferred back to main memory, the bound *worker* thread notifies the end of the execution to the *helper* thread, which can then continue with the notification of new ready tasks. The number of *worker* threads (an thus, the number of accelerators used in the executions) can be specified by the user at runtime.

4.3 Locality Exploitation and Task Scheduling

The two classes of available memory spaces (host and device memories) can be seen as a two-level memory hierarchy: before executing a task, the bound *worker* thread transfers the necessary blocks to the local memory of its GPU, performs the computations, and transfers back the updated data.

The impact of data transfers on the basic implementation explained above can be reduced by considering the local memory space of each accelerator as a cache memory that stores recently-used data blocks. The implementation of a software cache of read-only blocks stored in the memory of each GPU can reduce the amount of data transfers between host and device memory spaces. The replacement policy (LRU in our experiments) and the number and size of cache blocks can be tuned to improve performance.

In combination with the software cache, we have implemented two different memory coherence policies to reduce the amount of data transfers:

Write-invalidate: When the execution of a task is completed, the corresponding *worker* thread invalidates the read-only copies of the blocks modified by the active GPU on the memories of the remaining GPUs, notifying the *worker* threads bound to those GPUs to do so.

Write-back: To reduce the number of transfers further, a write-back policy is employed, allowing inconsistencies between data blocks stored in the caches of the accelerators and the blocks in RAM. Data blocks written by a GPU are only updated in RAM when another GPU has to operate with them.

The system automatically handles the existence of multiple memory spaces (as many as accelerators plus the system memory space) by keeping a memory map of the cache of each accelerator. The translation of addresses between memory spaces is transparent to the user, who is not aware of the existence of several separate memory spaces in his/her code. This centralized directory, managed by the *helper* thread, allows efficient handling of data blocks necessary for the management of the different data caches explained above. This software cache is an extension of that developed for CellSs, in which each *worker* thread had direct control over the bound SPE local store, being implemented in a distributed way.

Additionally, the information stored in the memory directory can be used to reduce data transfers by selectively mapping tasks to the most appropriate accelerator.

5 Experimental Results

In our experiments we used an NVIDIA TESLA s1070 computing system with four NVIDIA GT200 GPUs and 16 GBytes of DDR3 memory (4 GBytes per GPU). The TESLA system is connected to a workstation with two Intel Xeon QuadCore E5440 (2.83 GHz) processors with 8 GBytes of shared DDR2 RAM memory. The Intel 5400 chipset features two PCIExpress Gen2 interfaces connected with the TESLA, which deliver a peak bandwidth of 48 Gbits/second on each interface. NVIDIA `CUBLAS` (version 2.2) built on top of the CUDA API (version 2.2) together with NVIDIA driver (185.18) were used in our tests. MKL 10.0.1 was employed for all computations performed in the Intel Xeon using the 8 cores available in the system. Single precision was employed in all experiments. When reporting the rate of computation, we consider the cost of the Cholesky factorization to be the standard $n^3/3$ flops (floating-point arithmetic operations), for square matrices of order n. The GFLOPS rate is computed as the number of flops divided by $t \times 10^{-9}$, where t equals the elapsed time in seconds. The cost of all data transfers between RAM and GPU memories is included in the timings. No page-locked memory has been used in the allocation of the input matrices in the host memory.

Figure 3 (left side) shows the performance results for the Cholesky factorization algorithm shown in Figure 1, executed on the TESLA s1070 system using the GPUSs runtime. The four GPUs available in the TESLA system were used in the experiment, executing all the tasks exclusively on the graphics processor.

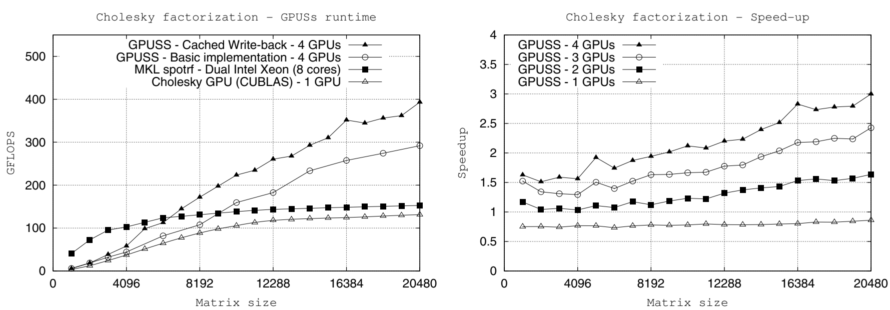

Fig. 3. Impact of the data cache and different memory coherence policies on the performance of the GPUSs runtime for the Cholesky factorization (left). Speed-ups of the runtime for the Cholesky factorization using 1, 2, 3, and 4 GPUs (right).

The optimal block size, experimentally determined, is much larger in our case than in the Cell B.E. implementation in [4]: observed optimal block sizes for the Tesla system were always equal or larger than 512, while in the CellSs case the optimal block size was 64 (limited by the small size of the Local Store of each SPE, and the existence of tuned versions of the kernels only for this block sizes).

An important improvement in performance is observed in Figure 3 when the write-back policy is incorporated. The figure also displays the performances of a LAPACK-like code linked with `CUBLAS` executed on a single GPU [3], and the MKL multi-threaded Cholesky factorization on the eight cores of the system.

The right-hand side plot in Figure 3 reports the speed-up of the algorithm in Figure 1, with the corresponding GPUSs annotations, and executed using the GPUSs runtime system on 1, 2, 3, and 4 processors of the Tesla system. Speed-ups are calculated with respect to the same algorithm, linked with CUBLAS, and run on a single processor of the Tesla. The results in the figure corresponding to a single processor reveal the small overhead introduced by the runtime.

6 Concluding Remarks

In this paper we have validated the versatility of the StarSs programming model, extending it to target architectures equipped with multiple hardware accelerators. With a very small number of modifications to user's code, our approach deals with data transfers, different memory spaces, and task scheduling in a heterogeneous system. The parallelization of the codes is performed by a runtime system based on the CellSs runtime, with notable preliminary performance results for a complex operation like the Cholesky factorization.

Although the experiments and runtime have been developed for a specific multi-GPU system (Nvidia Tesla), many of the ideas introduced here can also been applied to other architectures consisting of a workstation connected to multiple hardware accelerators via a fast interconnect.

Future work will include the implementation in the runtime of some of the extensions proposed in [2], mainly the possibility of deciding the target device (in our case, host or GPU) for the execution of a given task based on the state of the system. More complex scheduling strategies or software cache implementations, the implementation of multi-buffering to overlap transfers and computation, and ports to other multi-accelerator architectures are also in the roadmap.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. The researchers at UJI were supported by the Spanish Ministry of Science and Innovation/FEDER (contracts no. TIN2005-09037-C02-02 and TIN2008-06570-C04-01) and by the

Fundación Caixa-Castelló/Bancaixa (contracts no. P1B-2007-19 and P1B-2007-32). Part of this work was performed while Francisco D. Igual was visiting BSC-UPC. Support for this visit came from the *Spanish ICTS program* of the BSC.

References

1. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F.D., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A proposal to extend the OpenMP tasking model for heterogeneous architectures. In: Evolving OpenMP in an Age of Extreme Parallelism. 5th International Workshop on OpenMP, IWOMP 2009, Dresden, Germany. LNCS. Springer, Heidelberg (2009)
3. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 739–748. Springer, Heidelberg (2008)
4. Bellens, P., Pérez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 86. ACM Press, New York (2006)
5. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast matrix multiplication. IEEE Trans. on Parallel and Distributed Systems 13(11), 1105–1123 (2002)
6. Dongarra, J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. 16(1), 1–17 (1990)
7. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 101–110. ACM Press, New York (2009)
8. NVIDIA. NVIDIA CUDA Programming Guide 2.2 (2008)
9. Park, N., Hong, B., Prasanna, V.K.: Tiling, block data layout, and memory hierarchy performance. IEEE Trans. on Parallel and Distributed Systems 14(7), 640–654 (2003)
10. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the cell broadband engine processor. IBM Journal of Research and Development 51(5) (August 2007)
11. Perez, J.M., Badia, R.M., Labarta, J.: Scalar-aware grid superscalar. DAC TR UPC-DAC-RR-CAP-2006-12. Technical report, Universitat Politécnica de Catalunya, Computer Architecture Department (2006)
12. Pérez, J.M., Badia, R.M., Labarta, J.: A flexible and portable programming model for SMP and multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center - CNS, Barcelona, Spain (2007)
13. Quintana-Ortí, G., Igual, F.D., Quintana-Ortí, E.S., van de Geijn, R.A.: Solving dense linear systems on platforms with multiple hardware accelerators. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 121–130. ACM, New York (2009)