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

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

[1]Departamento de Ingeniería y Ciencia de los Computadores.
University Jaume I. Castellón (Spain)

[2]Barcelona Supercomputing Center - Centro Nacional de Supercomputación.
Barcelona (Spain)

Motivation (I)

The past: emergence of hardware accelerators

- Hardware accelerators (especially GPUs) have become a real solution for HPC
- Hardware: manycore systems on chip (up to 240 *cores* on modern Nvidia GPUs)
- Software: problem solved with high level programming and execution models (e.g. Nvidia CUDA, Brook+, OpenCL)

Motivation (II)

The present: heterogeneous multi-accelerator systems

- One accelerator is not always enough for many applications
- Different accelerators adapted to specific applications
- Multi-accelerator systems are the next step
- Hardware: Nvidia Tesla series, multiple ClearSpeed boards per system, hybrid architectures, ...
- Software: the problem is not solved yet:
  - Big code modifications from sequential code
  - Manual scheduling
  - The user has to know the best accelerator for each part of the application

Motivation (III)

The future: heterogeneous multi-accelerator systems (on-chip)

- Number of cores is increasing
- The programmability problem must be addressed as soon as possible
- Hardware: Larrabee, AMD Fusion, ...
- Software: will determine the success or failure of novel architectures

Outline

Contents

Introduction. StarSs

- The StarSs programming model addresses the programmability problem by exploiting *task level parallelism*
- It consists of:
  - A few OpenMP-like pragmas identifying tasks in the user code
  - A source-to-source compiler
  - A runtime system adapted to the underlying architecture
- Many instantiations of StarSs have been developed: CellSs, SMPSs, GridSs
- Each instantiation targets one specific architecture

Introduction. GPUSs

Our proposal: GPUSs

GPUSs: Instantation of the StarSs programming model focusing heterogeneous multi-accelerator platforms

—

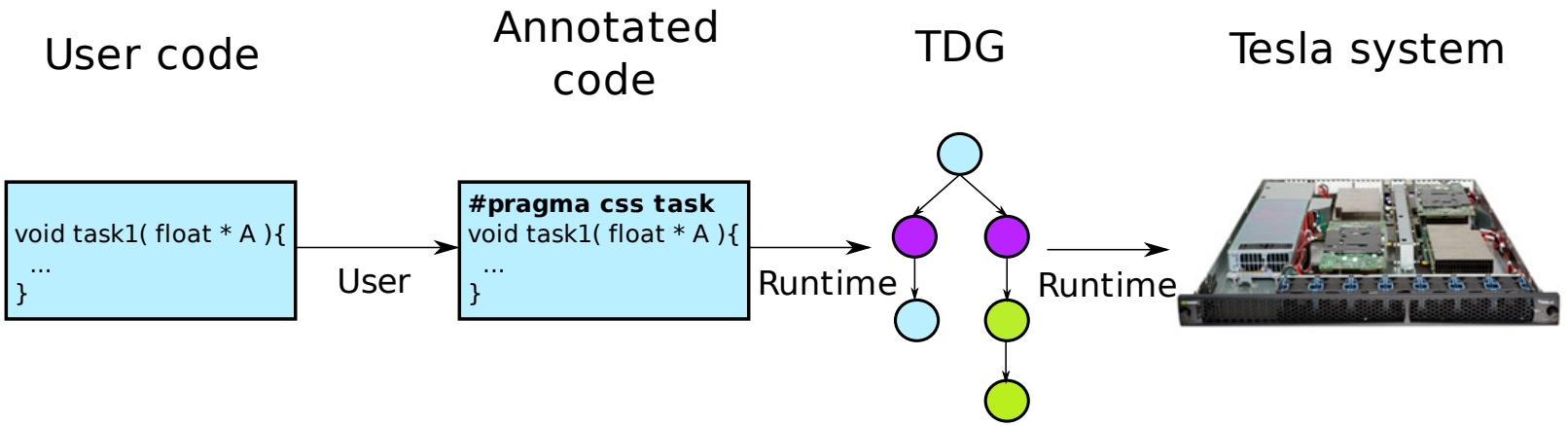
1. **Heterogeneity**: The target architecture is an heterogeneous multi-accelerator system
2. **Separate memory spaces**: The user does not have to deal with separate memory spaces for each accelerator
3. **Simplicity**: It adds few pragmas to the sequential user code to port it to the multi-accelerator system
4. **Portability**: It can be easily ported to other similar architectures based on multiple accelerators

Contents

The StarSs programming model

StarSs programming model

- **Automatic** parallelization of sequential applications
- **Runtime system**: efficient use available resources (e.g. GPUs) in parallel
- The user **annotates** the application: pieces of code that will be executed on a GPU (**tasks**)
- Runtime extracts parallelism building a **data dependency graph**



Proposed extensions

Extensions to the StarSs programming model

GPUSs provides OpenMP-like constructs to annotate code:

- To identify a unit of work, or task: **pragma css task**
- To select the execution device: **pragma css target device**

Defining tasks: the `task` clause

Taskifying functions

```
#pragma css task [clause_list]
 {function-header | function-definition}
```

- The `task` clause denotes a function that is always executed as a **task**.
- Whenever the program calls a function annotated in this way, the runtime will create an explicit task.

Defining tasks: the `task` clause

Identifying the directionality of the arguments

```
#pragma css task input( parameter ) |
                 output( parameter ) |
                 inout( parameter )
 {function-header | function-definition}
```

- The `input, output and inout` clauses denote the directionality of each argument.
- Used by the runtime to track dependencies among tasks and manage data transfers.

Specifying target devices: the `target` clause

Specifying target devices

```
#pragma css target device(device-name-list) [clause-list]
 {function-header|function-definition}
```

- The `target` construct specifies that the execution of a task can be offloaded on a given **device**.
- The target device is specified in `device-name-list`.
- When a task becomes ready, the runtime can choose among the available targets to decide where to execute the task.

Managing heterogeneity: the `implements` clause

- The `implements` clause is used to specify alternative implementations for a function

Example

```
#pragma css task
void matmul( float *A, float *B, float *C );

#pragma css target device( cuda ) implements( matmul )
void matmul_cuda( float *A, float *B, float *C ) {
  // tuned version for a CUDA-compatible device
}

#pragma css target device( smp ) implements( matmul )
void matmul_smp( float *A, float *B, float *C ) {
  // tuned vrsion for a SMP device
}
```

Example: the matrix-matrix multiplication

Parallelizing the matix-matrix multiplication

```
#pragma css task input(A[BS][BS], B[BS][BS]) inout (C[BS][BS])
#pragma css target device( cuda )
void matmul( float *A, float *B, float *C ) {
  // tuned CUDA code for the matmul
}

float *A[][], *B[][], *C[][];

int main( void ) {
  for( int i=0; i<NB; i++ )
    for( int j=0; j<NB; j++ )
      for( int k=0; k<NB; k++ )
        matmul( A[i][k], B[k][j], C[i][j] );
}
```

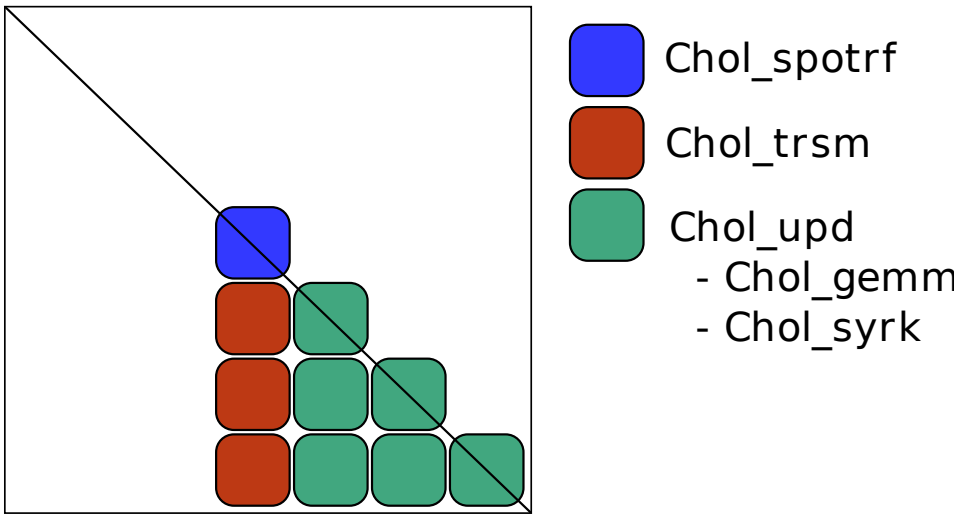
Example: the Cholesky factorization

The Chokesky factorization of a dense SPD matrix $A \in \Re^{n \times n}$ is defined as

$$A = LL^T$$

where $L \in \Re^{n \times n}$ is a lower triangular matrix.

Blocked algorithm:



Sequential Cholesky factorization

```
void Cholesky( float *A, int ts, int nt ) {
  for (int k = 0; k < nt; k++) {

    chol_spotrf( A[k*nt+k], ts );  // Factorize diagonal block

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

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

int main( void ) {
  float *A[nt][nt];
  ...
  // Compute the Cholesky factor
  Cholesky( A, ts, nt );
}
```

Taskifying the Cholesky factorization

Each block function can be converted into a **task**:

spotrf task

```
#pragma css task inout(A[NT][NT])
void chol_spotrf( float *A ) {
  spotrf( "Lower", &ts, A,
                   &ts, &info );
}
```

ssyrk task

```
#pragma css task input(A[NT][NT])
                 inout(C[NT][NT])
void chol_syrk( float *A,
                float *C ) {
  ssyrk( "L", "N", &ts, &ts,
         -1.0, A, &ts,
          1.0, C, &ts );
}
```

sgemm task

```
#pragma css task input(A[NT][NT],
                       B[NT][NT])
                 inout(C[NT][NT])
void chol_sgemm( float *A,
                 float *B,
                 float *C ) {
  sgemm( "N", "T", &ts, &ts, &ts,
         -1.0, A, &ts, B, &ts,
          1.0, C, &ts);
}
```

strsm task

```
#pragma css task input(T[NT][NT])
                 inout(B[NT][NT])
void chol_strsm( float *T,
                 float *B ) {
  dtrsm( "R", "L", "T", "N",
         &ts, &ts, 1.0, T, &ts,
                        B, &ts);
}
```

Taskifying the Cholesky factorization

Each block function can be converted into a **task**:

spotrf task

```
#pragma css task inout(A[NT][NT])
void chol_spotrf( float *A ) {
  spotrf( "Lower", &ts, A,
                   &ts, &info );
}
```

sgemm task

```
#pragma css task input(A[NT][NT],
                       B[NT][NT])
                inout(C[NT][NT])
void chol_sgemm( float *A,
                 float *B,
                 float *C ) {
  sgemm( "N", "T", &ts, &ts, &ts,
         -1.0, A, &ts, B, &ts,
          1.0, C, &ts);
}
```

ssyrk task

```
#pragma css task input(A[NT][NT])
                 inout(C[NT][NT])
void chol_syrk( float *A,
                float *C ) {
  ssyrk( "L", "N", &ts, &ts,
         -1.0, A, &ts,
          1.0, C, &ts );
}
```

strsm task

```
#pragma css task input(T[NT][NT])
                 inout(B[NT][NT])
void chol_strsm( float *T,
                 float *B ) {
  dtrsm( "R", "L", "T", "N",
         &ts, &ts, 1.0, T, &ts,
                        B, &ts);
}
```

Specifying the target device for each task

- By default, each task is executed on the SMP device unless the `target` clause is given.
- Example: `chol_spotrf` can be executed on a CUDA-capable device:

spotrf task on a CUDA-capable device

```
#pragma css task inout(A[NT][NT]) target device( cuda )
void chol_spotrf( float *A ) {
    // CUDA kernel for
    // the Cholesky factorization
}
```

Specifying multiple implementations for each task

- Multiple implementations for the `chol_spotrf` can be given:

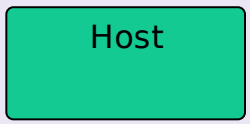
spotrf task on a CUDA-capable device

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

#pragma css task inout(A[NT][NT]) target device( cuda )
                 implements( chol_spotrf )
void chol_spotrf_cuda( float *A ) {
   // CUDA kernel for
   // the Cholesky factorization
}

#pragma css task inout(A[NT][NT]) target device( smp )
                 implements( chol_spotrf )
void chol_spotrf_smp( float *A ) {
   // SMP routine for
   // the Cholesky factorization
}
```

Contents

The target architecture

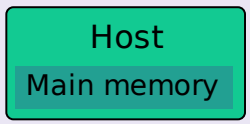
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

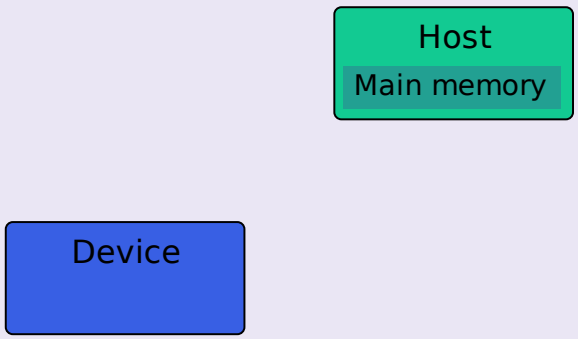
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

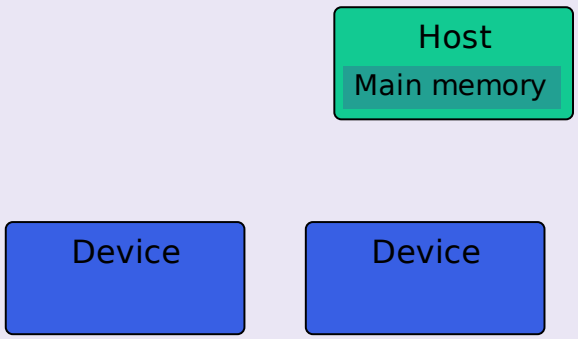
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

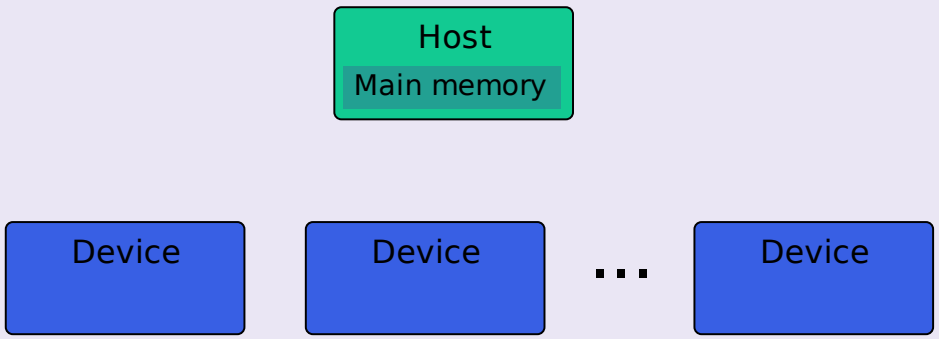
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

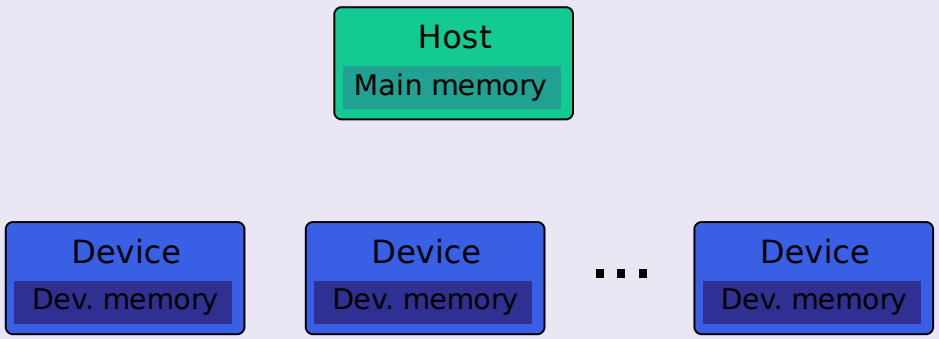
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

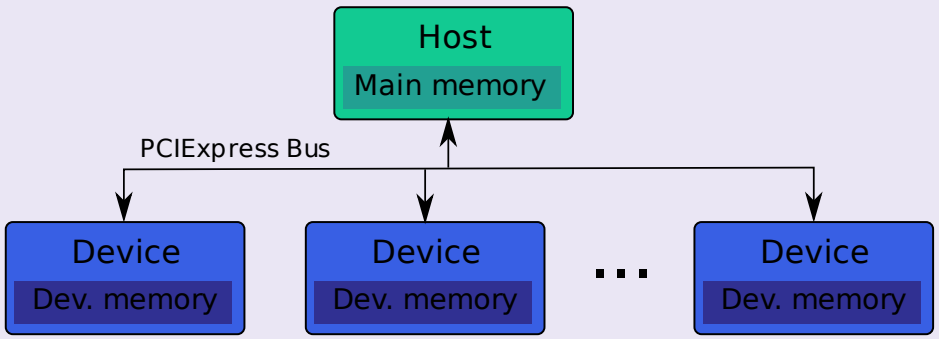
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

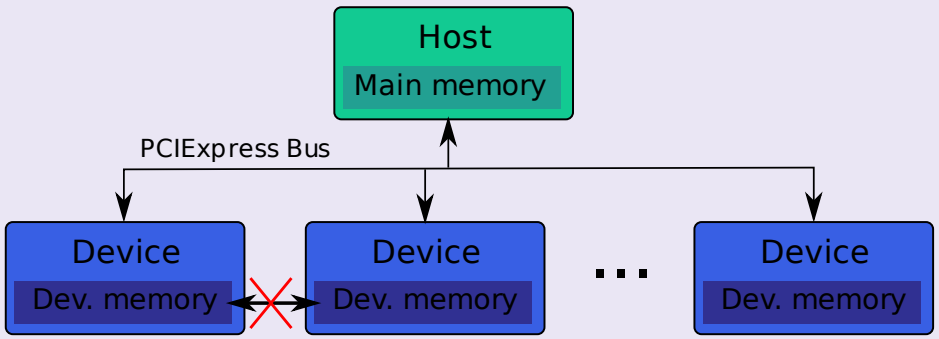
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

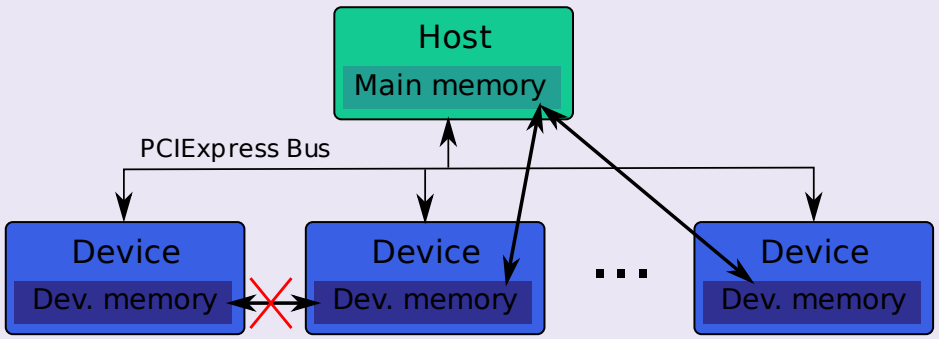
A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The target architecture

A typical multi-accelerator system



- **Host** with main memory
- **Devices** with local memory
- Communication through PCIExpress
- No direct device-device communication
- Communication through main memory

The GPUSs runtime. Overview

- Many features inherited from the CellSs and SMPSs runtimes
- Two main modules:
  1. Execution of the annotated user code, task generation and scheduling
  2. Data movements and task execution

The GPUSs runtime. Structure

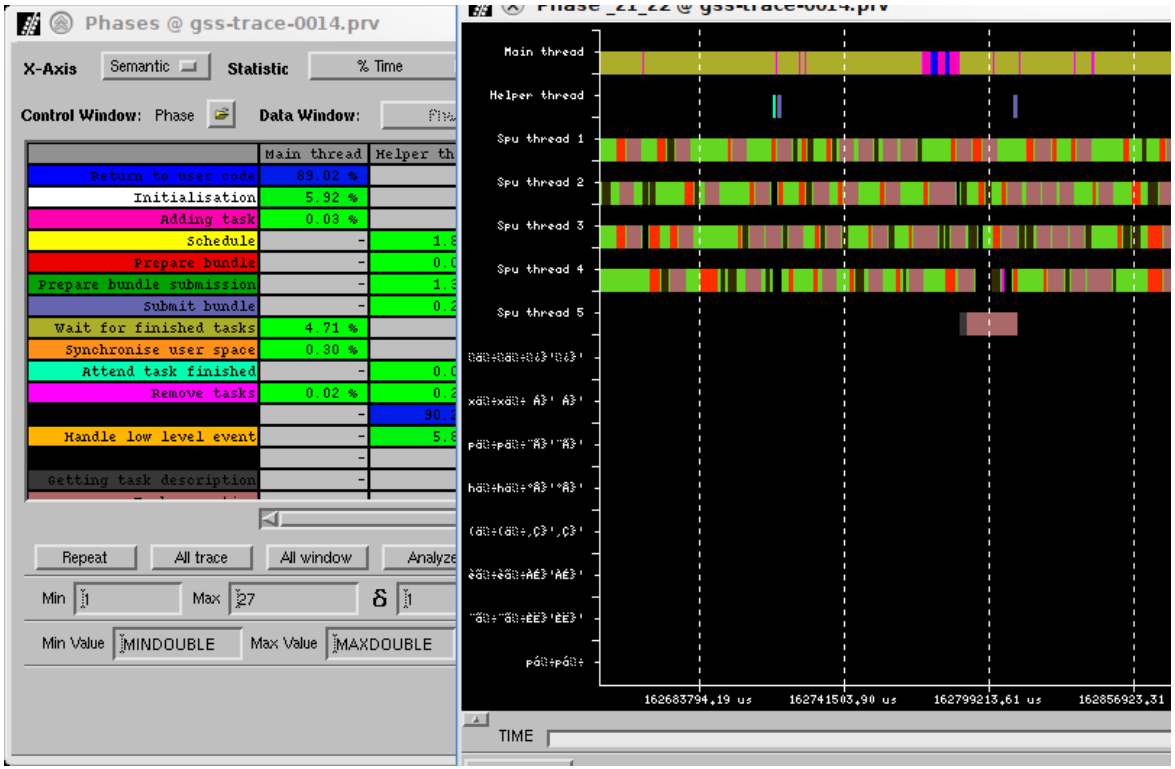
1. A *master* thread:
   - Executes the user code
   - Intercepts calls to annotated functions
   - Generates tasks
   - Inserts them in a **Task Dependency Graph**
2. A *helper* thread:
   - Consumes tasks from the TDG as the GPUs become idle
   - Maps tasks to the most suitable device
   - Intercepts finalization signals from the worker threads
3. A set of *worker* threads:
   - Wait for available tasks
   - Perform the necessary data transfers from RAM to GPU
   - Invoke the task call on the GPU
   - Retrieve the results (if necessary)

The GPUSs runtime. Locality exploitation

- Host and device memories: two-level memory hierarchy
  - Data is transferred to device memory prior to any task execution
  - Data is transferred back after execution
- Consider the local memory of each GPU as a cache memory storing recently-used data blocks
- Software cache + Memory coherence policies:
  - Write-invalidate
  - Write-back
- The runtime keeps a memory map of each accelerator cache
- This information can be used to improve the mapping of tasks to resources

The GPUSs runtime. Additional features

- Definition of the number of accelerators at runtime
- *Paraver* traces to analyze performance



- Hybrid CPU/GPU execution of tasks
- Ported to a system with multiple ClearSpeed boards

Contents

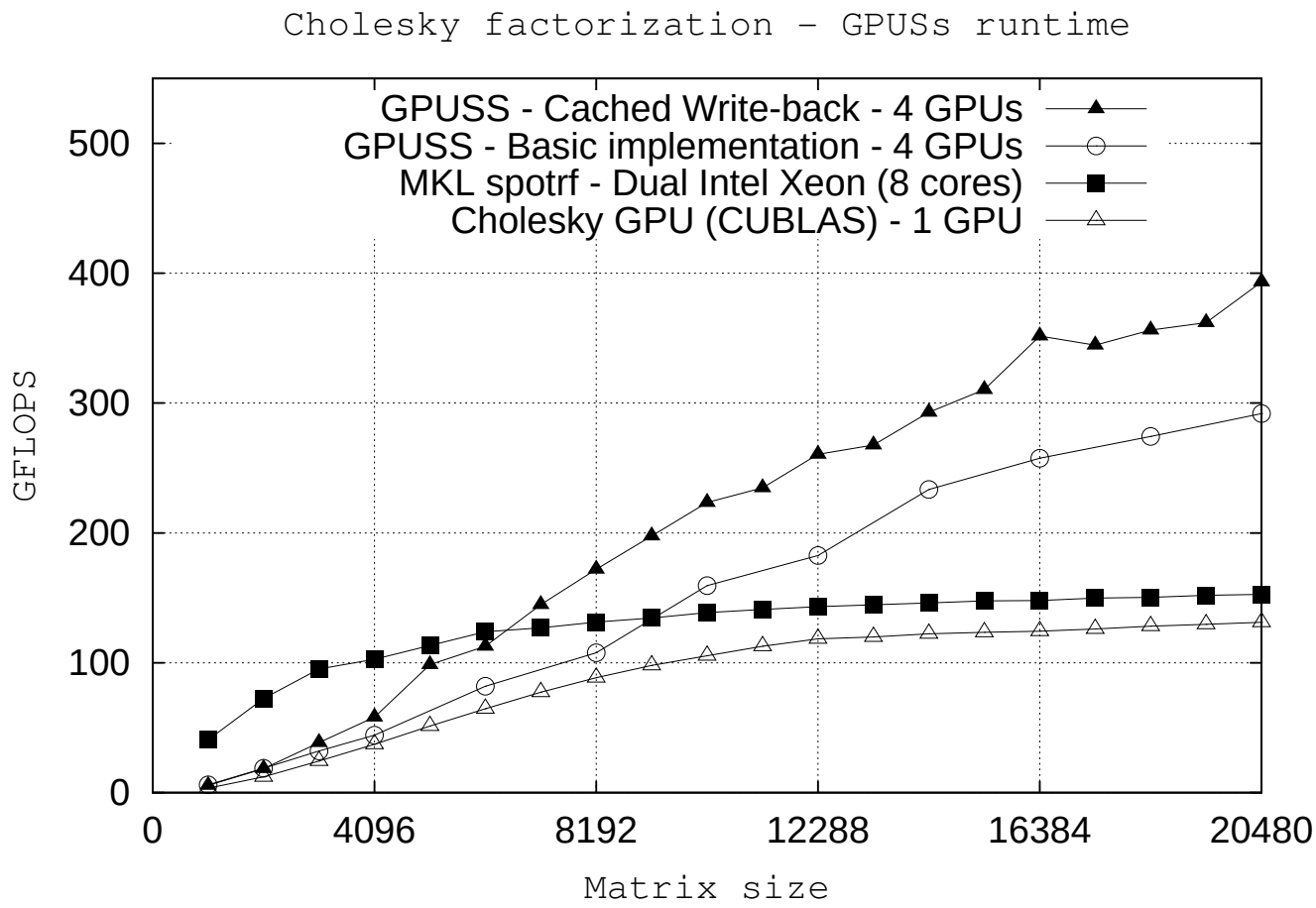
Experimental results

Experimental setup

CPU	Dual Xeon QuadCore E5440
CPU frequency	2.83 Ghz
RAM memory	16 Gbytes
GPU	Tesla s1070
Graphics processors	4 x GT200
GPU frequency	1.3 Ghz
Video memory	4 Gbytes per GPU
Interconnection	PCIExpress Gen2
CUDA version	2.0
MKL version	10.0.1
Driver version	185.18

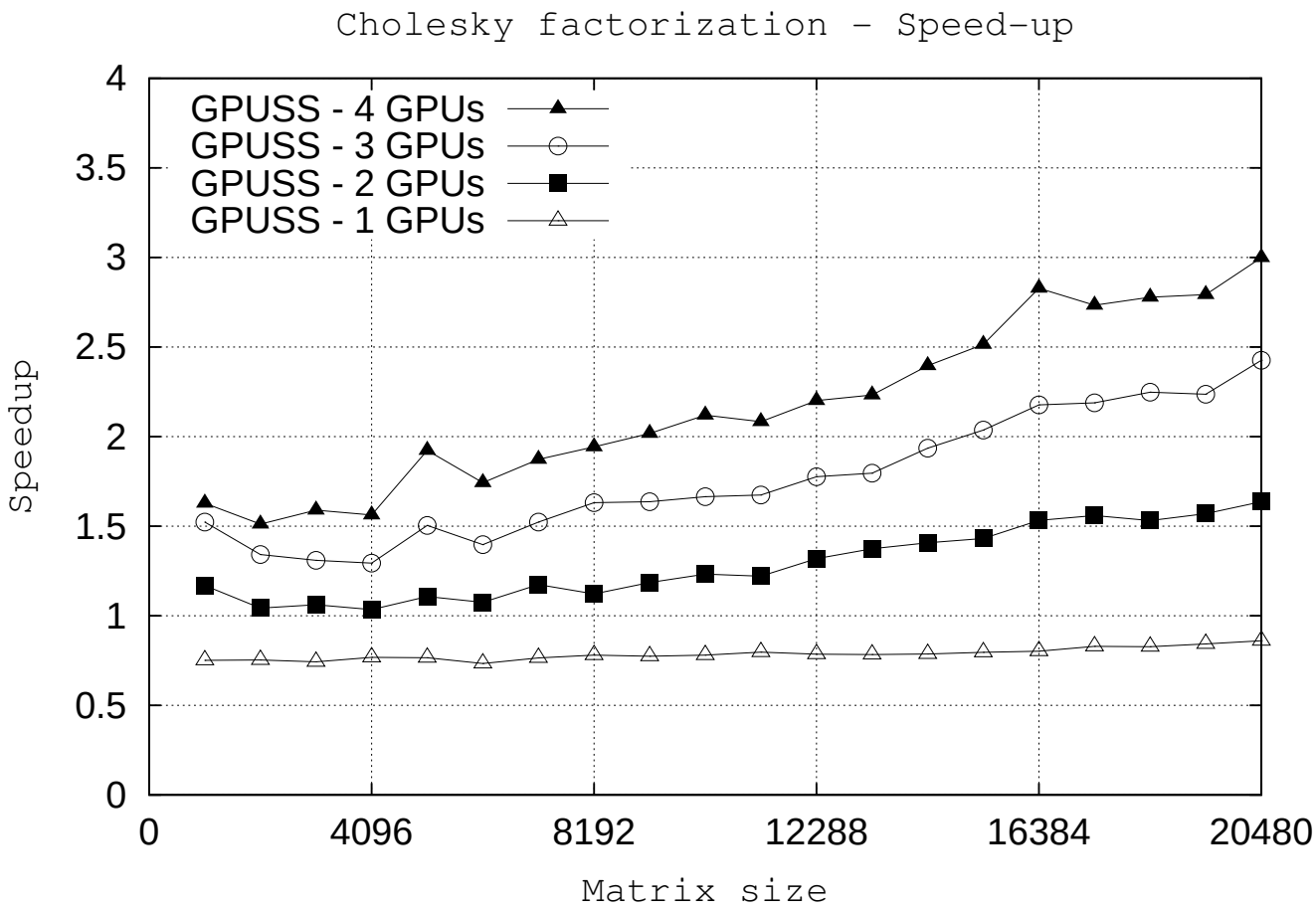
Performance measured in GFLOPS

Experimental results. Cholesky factorization



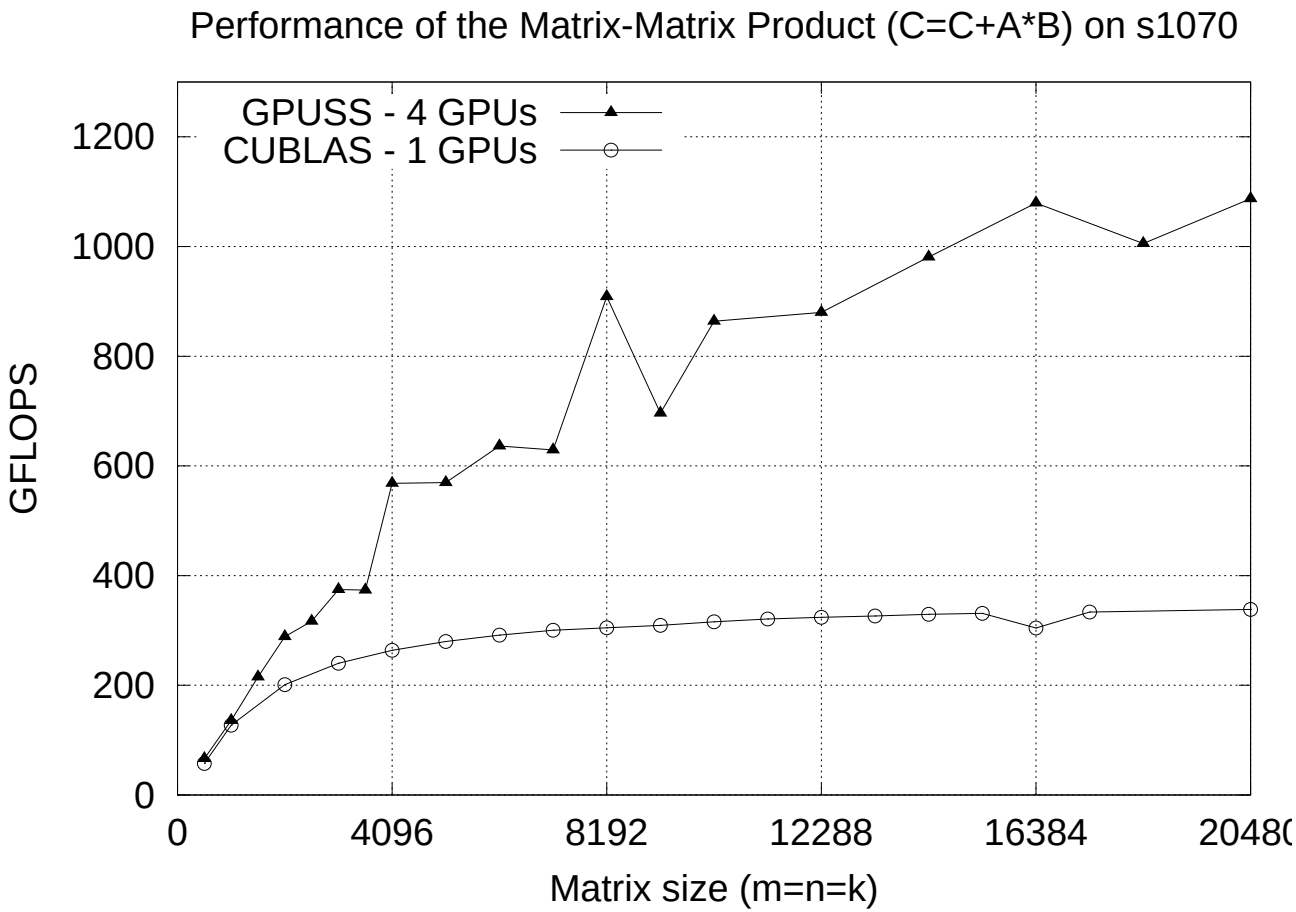
- Tasks executed exclusively on GPUs (simple precision)
- Important improvement with software cache

Experimental results. Scalability



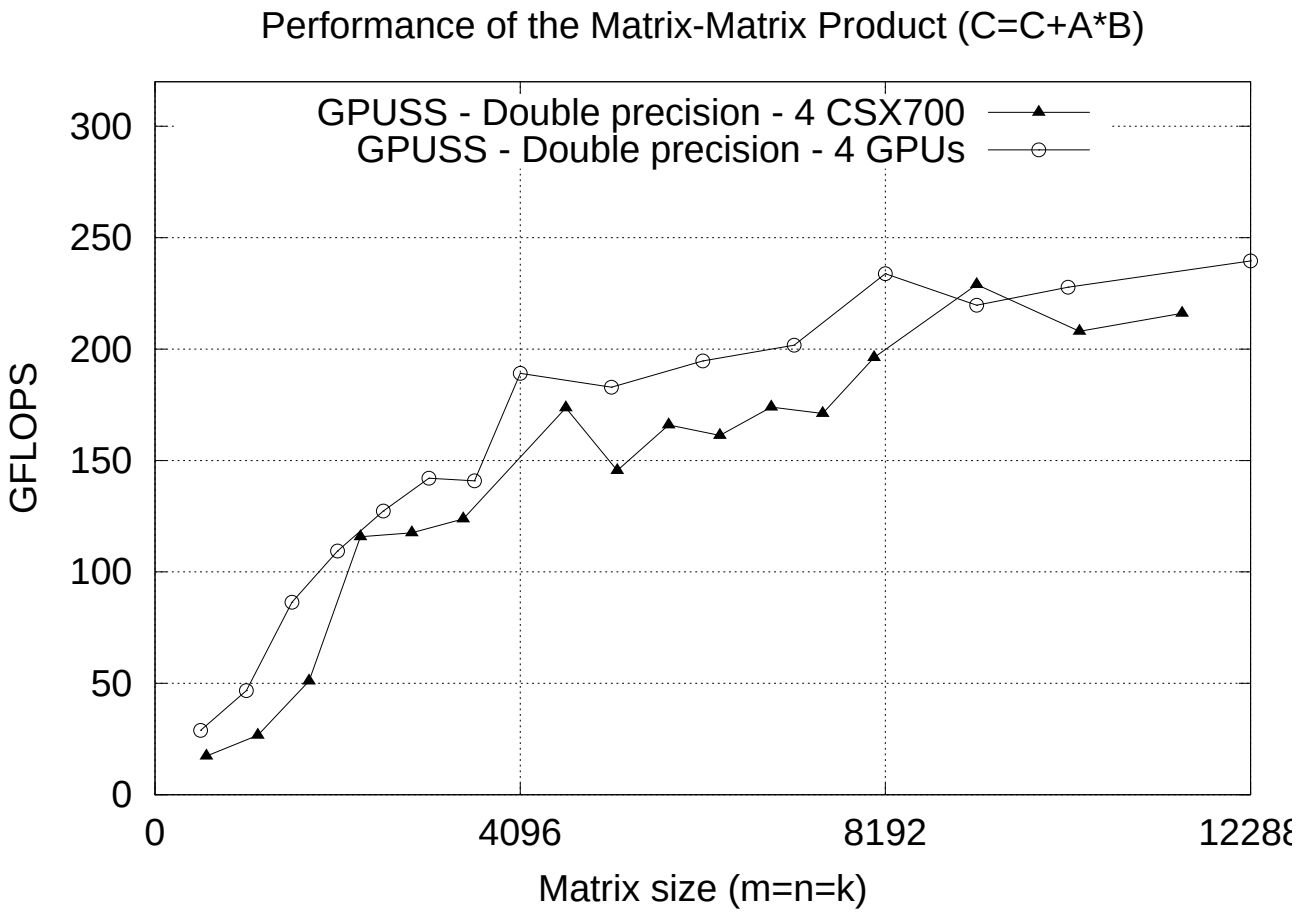
- PCIExpress: main bottleneck as number of GPUs increases

Experimental results. GEMM



- Performance near 1.1 TFlop (346 GFlops on 1 GPU using CUBLAS)

Experimental results on ClearSpeed boards



- Performance similar to 4 GPUs (double precision)

Contents

Conclusions

Conclusions

- StarSs programming model: versatile and extensible for new architectures
- Programmability will determine the success of emerging architectures
- Our approach relies on a runtime system: little user intervention
- Many ideas can be applied to other multi-accelerator systems

Future work

- More complex scheduling strategies
- Porting to other multi-accelerator platforms
- Porting to heterogeneous multi-accelerator platforms
- Let the runtime automatically decide where to execute each task

Conclusions

Conclusions

- StarSs programming model: versatile and extensible for new architectures
- Programmability will determine the success of emerging architectures
- Our approach relies on a runtime system: little user intervention
- Many ideas can be applied to other multi-accelerator systems

Future work

- More complex scheduling strategies
- Porting to other multi-accelerator platforms
- Porting to heterogeneous multi-accelerator platforms
- Let the runtime automatically decide where to execute each task

Questions?

Related work

Related work

SuperMatrix
- Extension of the SuperMatrix SMP runtime
- Automatic parallelization of linear algebra programs
- Hybrid CPU / Multi-GPU systems

Volkov et al.
- Some highly tuned codes for multi-GPU systems
- Linear algebra codes
- No runtime or automatic scheduling

Lee et al.
- Compiler framework for automatic translation and optimization
- OpenMP $\rightarrow$ GPU translation

StarPU

Related work

Related work

SuperMatrix
- Extension of the SuperMatrix SMP runtime
- Automatic parallelization of linear algebra programs
- Hybrid CPU / Multi-GPU systems

Volkov et al.
- Some highly tuned codes for multi-GPU systems
- Linear algebra codes
- No runtime or automatic scheduling

Lee et al.
- Compiler framework for automatic translation and optimization
- OpenMP $\rightarrow$ GPU translation

StarPU

Related work

Related work

SuperMatrix
- Extension of the SuperMatrix SMP runtime
- Automatic parallelization of linear algebra programs
- Hybrid CPU / Multi-GPU systems

Volkov et al.
- Some highly tuned codes for multi-GPU systems
- Linear algebra codes
- No runtime or automatic scheduling

Lee et al.
- Compiler framework for automatic translation and optimization
- OpenMP $\rightarrow$ GPU translation

StarPU

Related work

Related work

SuperMatrix
- Extension of the SuperMatrix SMP runtime
- Automatic parallelization of linear algebra programs
- Hybrid CPU / Multi-GPU systems

Volkov et al.
- Some highly tuned codes for multi-GPU systems
- Linear algebra codes
- No runtime or automatic scheduling

Lee et al.
- Compiler framework for automatic translation and optimization
- OpenMP $\rightarrow$ GPU translation

StarPU

Tesla vs. Cell B.E.

Similarities with the Cell B.E.

- Heterogeneous architectures:
  - Cell B.E.: 1 PPE + 8 SPEs
  - Tesla: 1 (multicore) CPU + 4 GPUs
- Each accelerator has its own local memory pool
- Fast interconnection network

Differences with the Cell B.E.

- GPUs need more granularity to attain good performance
- PPE performance is poor compared to that of the SPE
- Larger local memory spaces for each GPU (Gbytes) than for each SPE (Kbytes)
- Impact of data transfers (PCIExpress vs EIB)
- GPUs are passive elements: no system threads can be run on them

Specifying data movements

Some additional clauses can be used with the `device` pragma:

Data movement clauses

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

These clauses specify data movement for the shared variables inside a task:

- `copy_in` moves variables from host to device memory once the task is ready for execution.
- `copy_out` moves variables from device to host once the task finishes execution.