Perez, Bellens, Badia, and Labarta

“CellSs Making it easier to program the Cell Broadband Engine processor”

Presented by:
Mujahed Eleyat

Outline

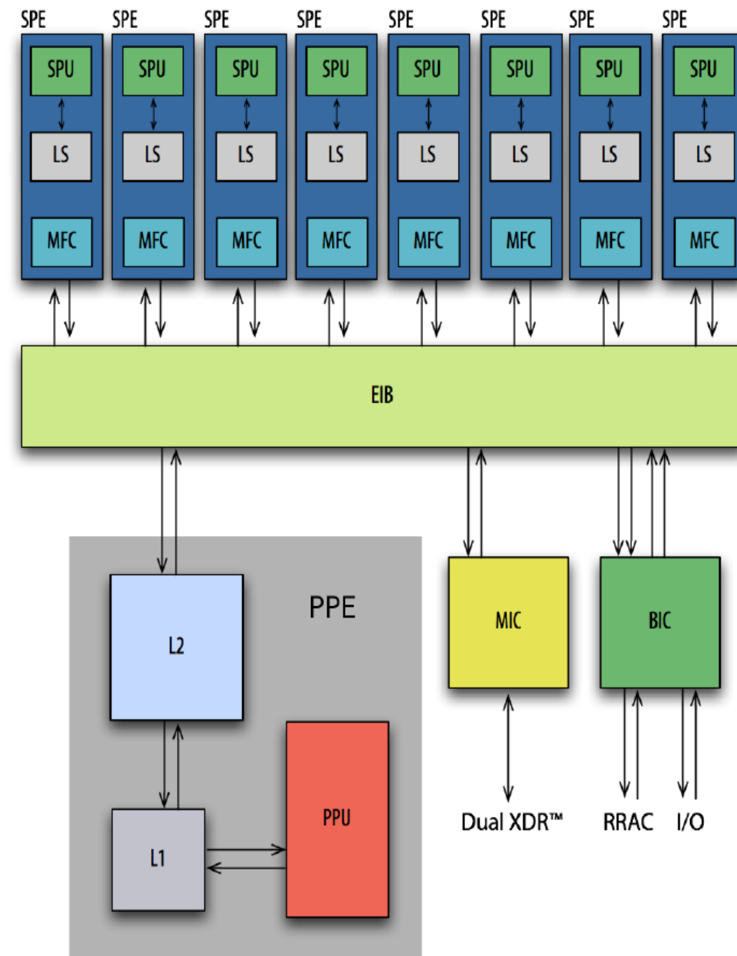
- Motivation
- Architecture of the cell processor
- Challenges of Cell programming
- Cell Superscalar
- Pragma types
- CellSs compiler
- CellSs runtime behavior
- Reducing scheduling overhead
- Tracing, Scalability, and Performance

Motivation

- Current industry has moved to multicore designs.
- Increasing the performance of single core processors become difficult due to power consumption and heat dissipation
- Recent Berkley report mentioned that Current programming methodologies can't be used for more than 16 cores
- Intel research prototype (80 core)
- Current methods should be shifted to
  - Maximize human productivity
  - Programming models independent of the cores count
  - Wide range of data types should be supported
  - Task-, word-, and bit-level parallelism

Architecture of the Cell Processor

- 1 PPE (Power Processing Element)
  - 64-bit PowerPC
  - 32 KB Instr/Data L1 Cache, 512 KB L2 Cache
  - SMP (2 threads)
  - 3.2 GHz (SP: 25.6 GFlops DP: 6.4 GFlops)
- 8 SPEs (Synergistic Processing Element)
  - 256 KB Local Store
  - 3.2 GHz (SP: 25.6 GFlops DP: 1.83 GFlops)
  - 128-bit Vector Registers
- EIB (Element Interconnect Bus)
  - Interconnects PPE, SPEs, Memory, I/O
  - Simultaneous Read/Write
- MIC (Memory Interface Controller)
  - Interfaces to XDR Memory
  - Theoritical B/W of 25.6 GB/s



Challenges of Cell Programming

- Local store is small (256 KB) and not coherent with main memory
- To execute on SPE, Data is needed to be transferred from main memory to local store using DMA.
- Efficient execution on SPE requires using vectorized code using single precision data.

Cell Superscalar (CellSs)

- Proposed as a programming model for the multicore
- Currently focused on the Cell BE.
- Based on simple annotations (pragmas) to a sequential code.
  - Identify independent parts of the code (tasks) without collateral effects
- A source-to-source compiler is used to generate the code for both the PPE and SPEs
- Task dependency graph is built at runtime
- The runtime system tries to concurrently execute tasks without data dependency on different SPEs
- Data transfer between main memory and LS is handled by the system

Pragma types

- Initialization and finalization pragmas
  - `ccs start` and `css finish`
  - Indicate the beginning and end CellSs applications
- Task pragmas
  - Before some functions
  - Specify size of arrays and matrices
  - Specify direction of parameters (input, output, and inout)
- Synchronization pragmas
  - `css wait` is needed to access data generated by annotated functions

Example – Sparse LU

```
float *A[NB][NB];

#pragma css task inout(diag[B][B])
void lu0(float *diag){
   int i, j, k;

   for (k=0; k<BS; k++)
      for (i=k+1; i<BS; i++) {
         diag[i][k] = diag[i][k] / diag[k][k];
         for (j=k+1; j<BS; j++)
             diag[i][j] -= diag[i][k] * diag[k][j];
      }
}
#pragma css task input(diag[B][B]) inout(row[B][B])
void bdiv(float *diag, float *row){
...
}
#pragma css task input(row[B][B],col[B][B]) inout(inner[B][B])
void bmod(float *row, float *col, float *inner){
...
}
#pragma css task input(diag[B][B]) inout(col[B][B])
void fwd(float *diag, float *col){
...
}

void
write_matrix (FILE * file, float *matrix[NB][NB])
{
  int rows, columns;
  int i, j, ii, jj;

  fprintf (file, "%d\n %d\n", NB * B, NB * B);
```

```
  for (i = 0; i < NB; i++)
    for (ii = 0; ii < B; ii++)
      {
        for (j = 0; j < NB; j++){
#pragma css wait on(matrix[i][j])
          for (jj = 0; jj < B; jj++)
            fprintf (file, "%f ", matrix[i][j][ii][jj]);
        }
        fprintf (file, "\n");
      }

}

int main(int argc, char **argv) {
int ii, jj, kk;
FILE *fileC;
…
initialize (A);
#pragma css start
 for (kk=0; kk<NB; kk++) {
         lu0(A[kk][kk]);
      for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
                        fwd(A[kk][kk], A[kk][jj]);
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL) {
          bdiv (A[kk][kk], A[ii][kk]);
            for (jj=kk+1; jj<NB; jj++)
              if (A[kk][jj] != NULL) {
                 if (A[ii][jj]==NULL)
                   A[ii][jj]=allocate_clean_block();
                 bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
               }
                    }
    }

  fileC = fopen (argv[3], "w");
  write_matrix (fileC, A);
  fclose (fileC);
  #pragma css finish
}
```

CellSs Compiler

- A Source-to-source
- Generates two files:
  - The main code: to be executed by PPE
  - The task code: to be run by the SPEs
- In the beginning of the main code, the compiler inserts the following application calls to CellSs runtime
  - Calls to initializing and finalizing functions
  - Calls for registering annotated functions
  - Calls to a CellSs runtime library primitive (`css_addTask`) wherever a call to one of the annotated functions is found
- An adapter function for each annotated function is generated
- These adapters are called from the task main code
- The two files are compiled with the GCC or the ibm xlc

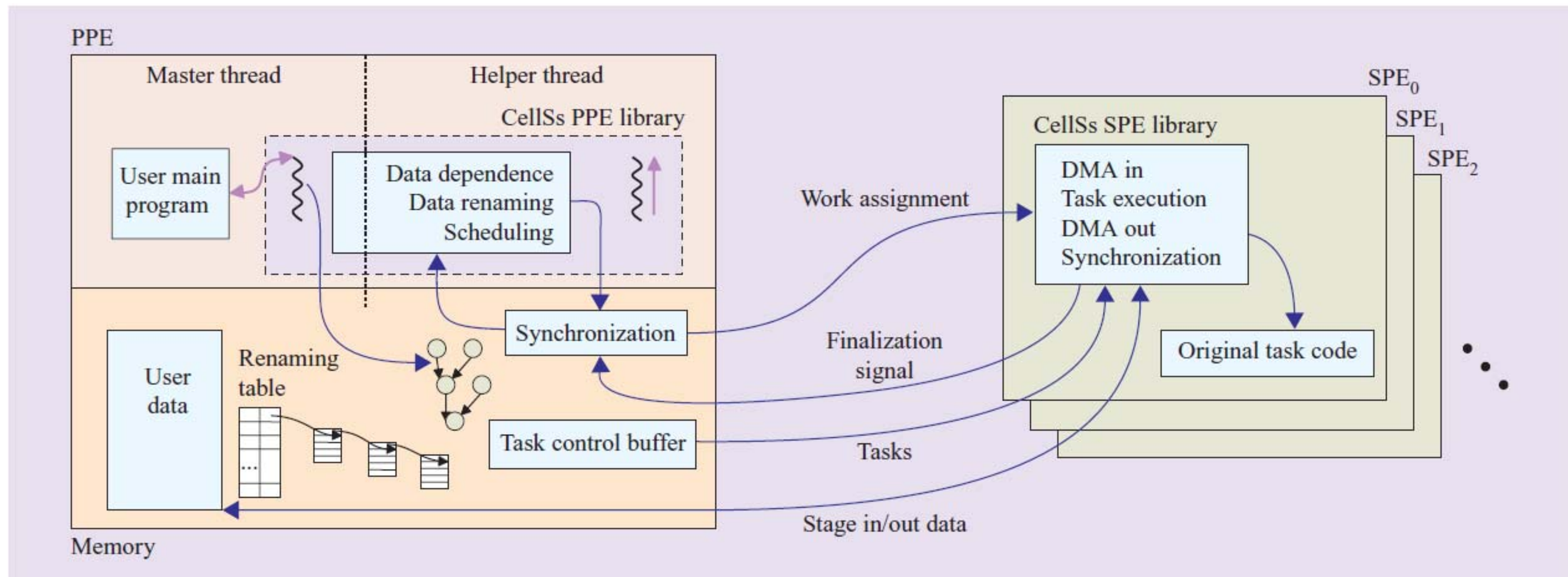
CellSs runtime behavior - 1

- CellSs application is called
- The main thread (master thread)runs on the PPE
- An additional thread (helper thread) also runs on the PPE
- In the initialization phase, the CellSs runtime starts as threads in the SPEs and start waiting for requests
- The master thread adds a task to a task graph whenever it calls the `css_addTask`
- An edge is added between tasks that have data dependency
- The master thread performs parameters renaming to remove WAW (write after write) and WAR (write after read)

CellSs runtime behavior - 2

- Tasks with no data dependency can be scheduled by the helper thread to execute on SPEs
- When the task finishes, the task program notifies the helper thread which then
  - Updates the task graph
  - Schedule new tasks for execution in idle SPEs
- A data structure, called the task control buffer, stores all the infrormation required by the SPE:
  - Task type identifier
  - Initial address and size of each parameter

CellSs runtime



Reducing scheduling overhead

- Task are grouped into bundles that are scheduled to one SPE
- To group the tasks, the following heuristics are followed:
  - Tasks forming a chain are grouped (one predecessor and one successor)
    - Reduces data transfer
  - Totally independent tasks are grouped
  - For more general data dependence organizations, the scheduler tries to form chains
- To reduce the execution time of a bundle of tasks, double buffering is implemented

Scheduling heuristics

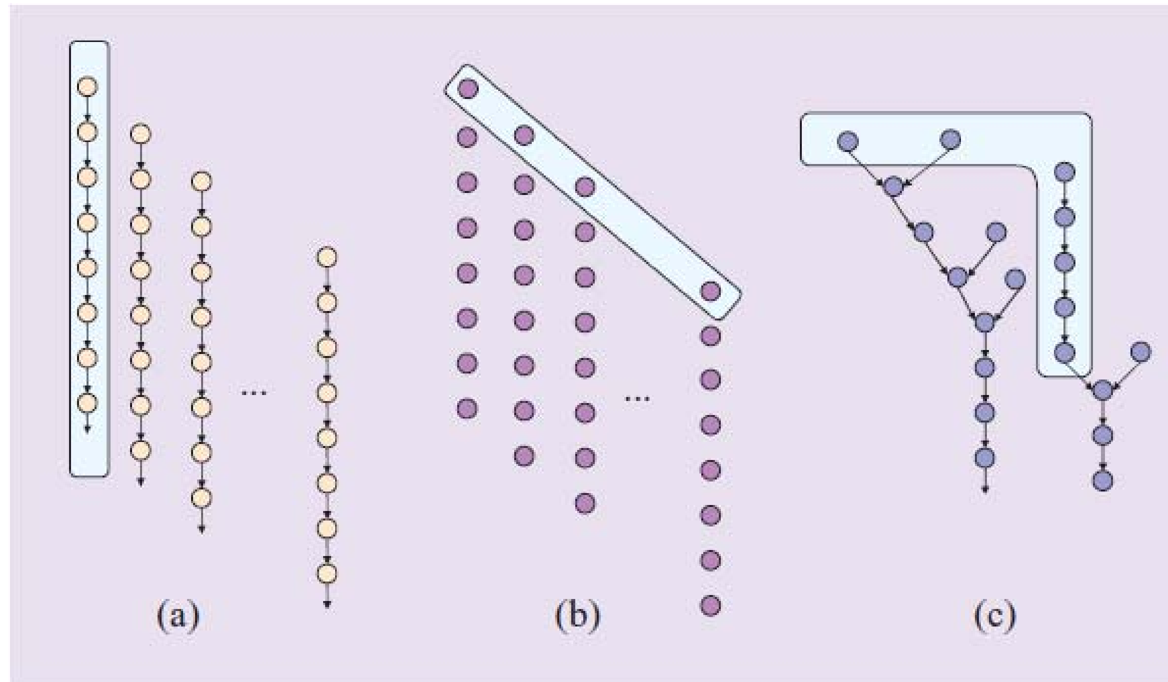


Figure 3

Sample tasks graphs: (a) tasks organized in chains; (b) totally independent tasks; (c) general dependency.

Double buffering in a task bundle

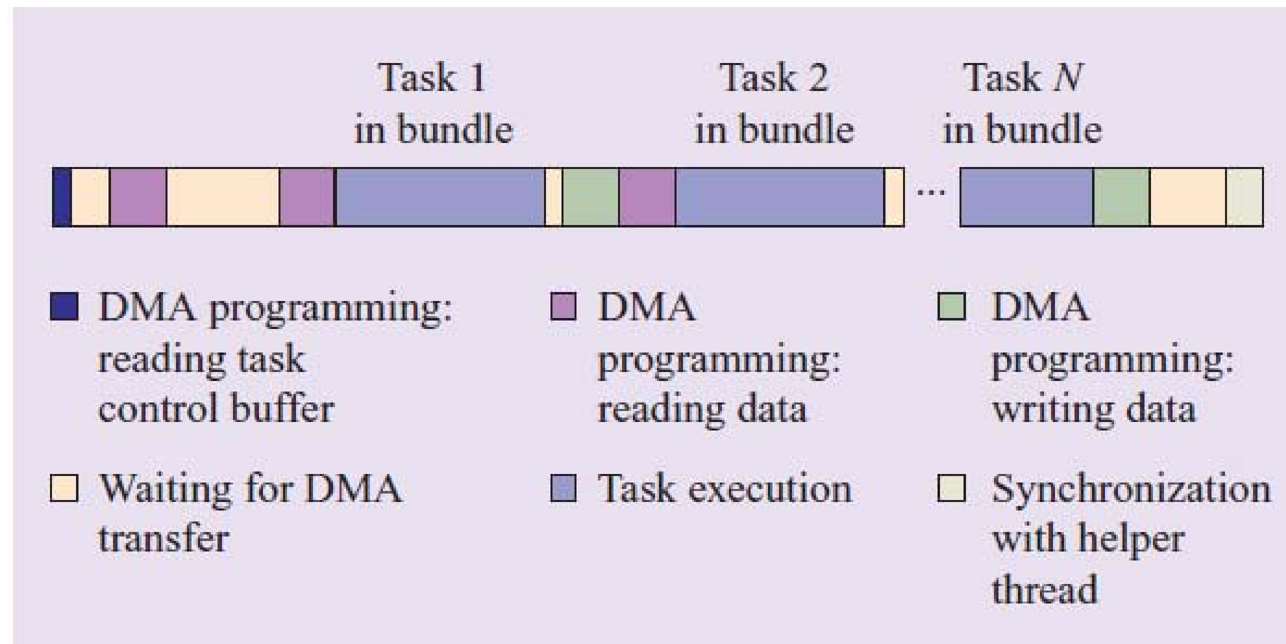


Figure 4

Double buffering in a task bundle.

Tracing

- A tracing component has been embedded in the CellSs runtime
- It generates postmortem trace files of the application
  - when the main program enters or exits any function of the CellSs library
  - when an annotated function is called in the main program
  - when a task is started or finished
- These traces can then be analyzed with the Paraver graphical user interface which allows doing performance analysis

Scalability analysis

Block matrix multiplication

- Each block 64 x 64 floats
- 16 x 16 blocks

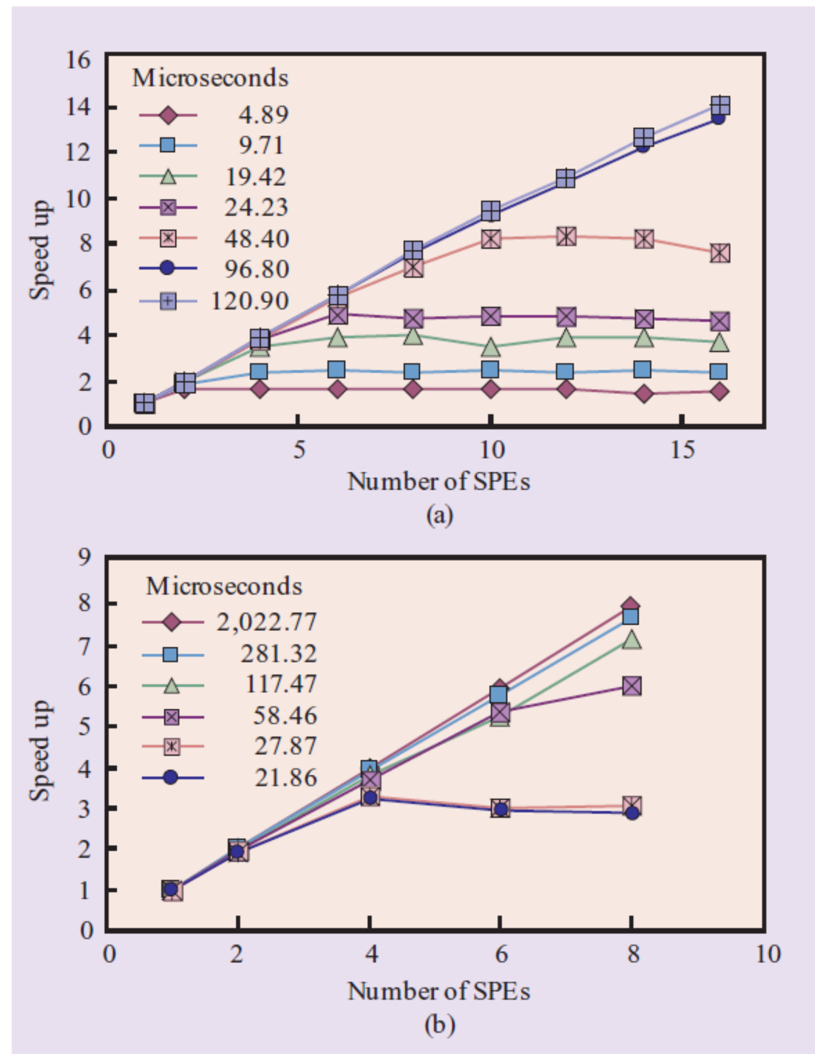


Figure 5

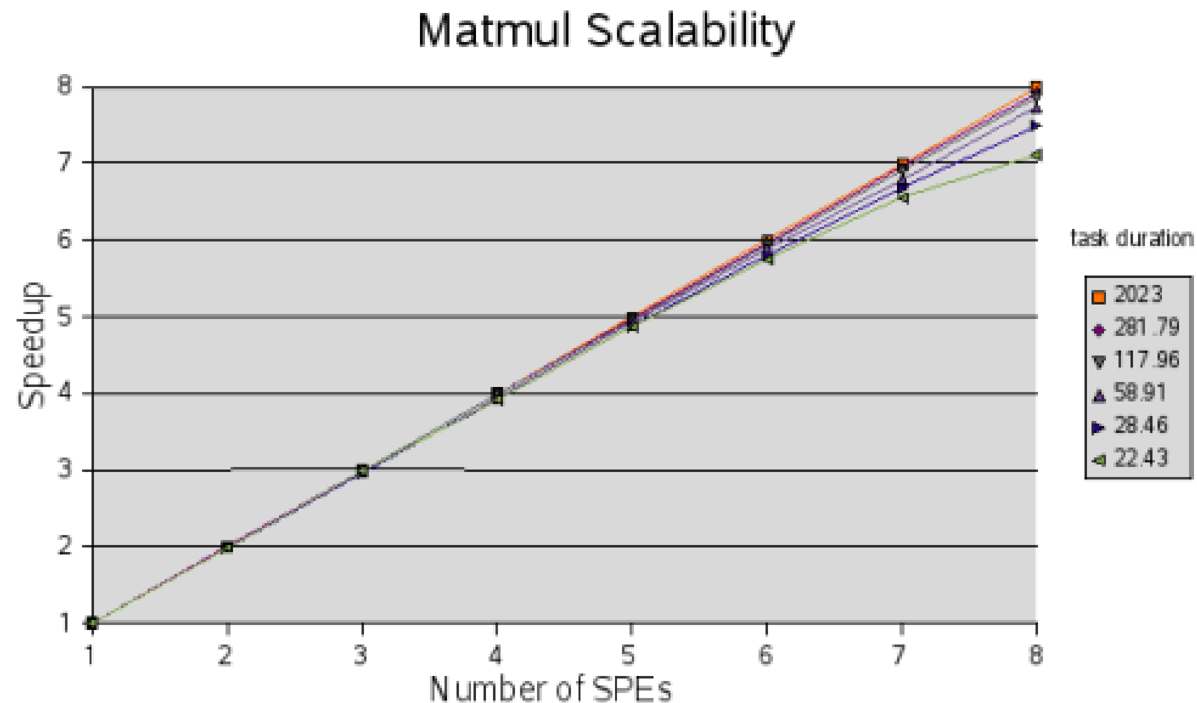
Scalability analysis of CellSs (a) with different task sizes and (b) of matrix multiplication.

Table 1 Performance results for the matrix multiplication (Gflops).

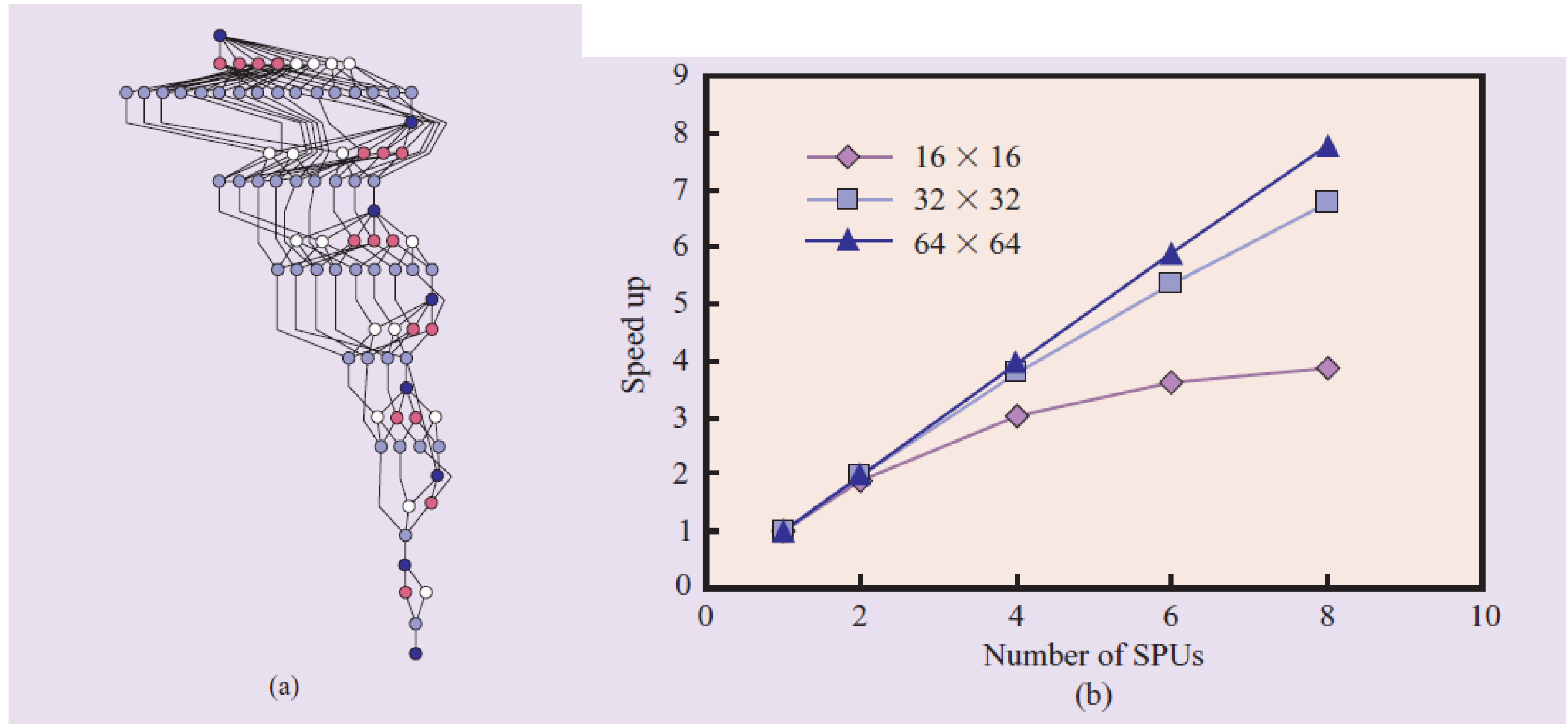
No. of SPEs	*Task size (μs)*			
	117.47	*58.46*	*27.87*	*21.86*
1	4.29	8.16	19.46	20.48
2	8.34	16.27	37.24	39.24
4	16.46	30.08	63.38	66.36
6	21.28	44.02	58.74	59.81
8	30.81	48.32	59.50	58.88

Performance results

- In the paper,
  - The higher performance is obtained with the kernel of the SDK and four SPEs (66.36 Gflops)
  - The reason for this behavior is that the master thread is busy adding tasks to the task dependency graph most of its time (88% of the time)
- The Barcelona website shows better results!

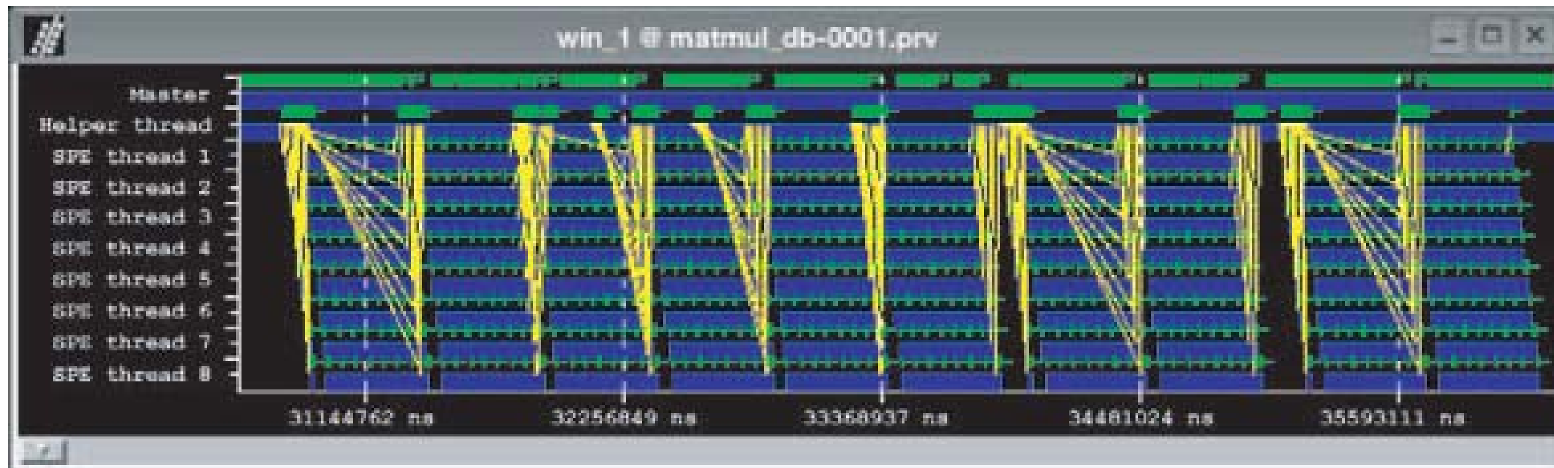


Sparse LU



Sparse LU example: (a) task-dependence graph (8 × 8 matrix case); (b) speed up for the sparse LU case for different matrices.

Performance analysis using Paraver - 1

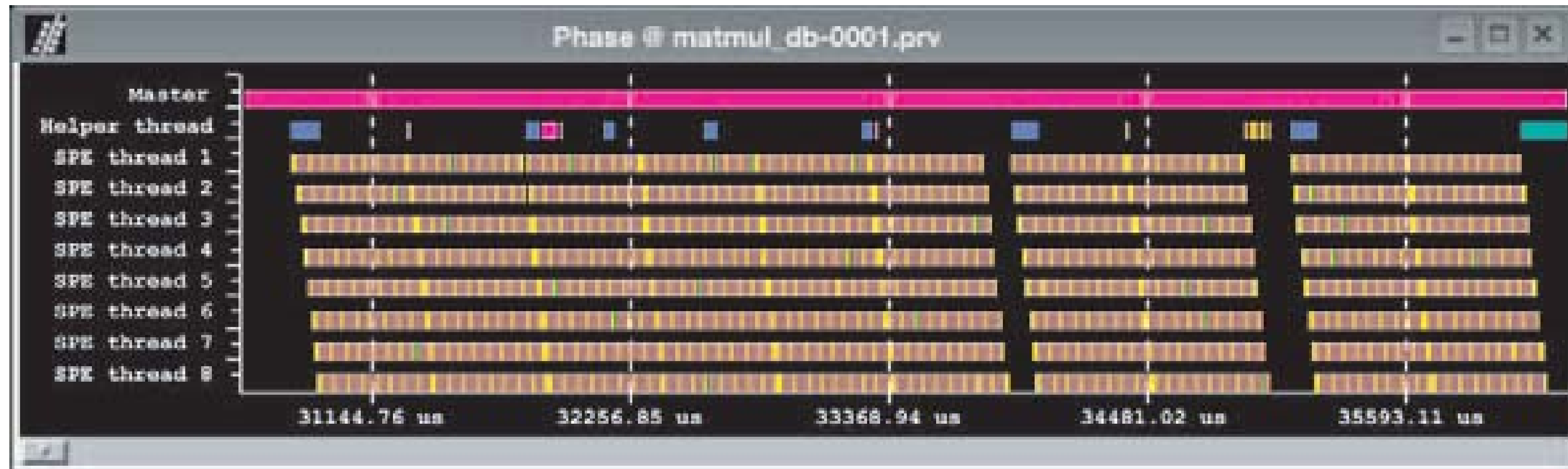


Green flags: events

Yellow lines: communications between SPE threads and the helper thread

Blue: activity

Performance analysis using Paraver - 2



Another view of Paraver

Yellow : DMA transfer (SPE)

Brown: SPE executing

Table 2 Data presented in Paraver view of the time invested in each phase by each thread.

	Master (%)	*Helper thread* (%)	*SPE thread 1* (%)	*SPE thread 2* (%)	*SPE thread 3* (%)	*SPE thread 4* (%)
Return to user code	4.73	—	—	—	—	—
Adding task	69.51	—	—	—	—	—
Schedule	—	7.36	—	—	—	—
Prepare bundle	—	2.56	—	—	—	—
Prepare bundle submission	—	0.76	—	—	—	—
Submit bundle	—	1.97	—	—	—	—
Attend task finished	—	4.09	—	—	—	—
Remove tasks	25.76	12.51	—	—	—	—
Low-level wait for events	—	70.76	—	—	—	—
Waiting for tasks	—	—	4.20	4.07	3.94	4.03
Getting task description	—	—	0.09	0.08	0.09	0.09
Task stage in	—	—	0.90	0.90	0.88	0.88
Task arguments alignment	—	—	0.27	0.25	0.27	0.28
Task execution	—	—	91.31	91.33	91.30	91.30
Task stage out	—	—	0.43	0.42	0.44	0.44
Task finished notification	—	—	0.05	0.05	0.05	0.05
Wait for DMA	—	—	2.76	2.89	3.03	2.94
Total	100.00	100.00	100.00	100.00	100.00	100.00
Average	33.33	14.29	12.50	12.50	12.50	12.50
Maximum	69.51	70.76	91.31	91.33	91.30	91.30
Minimum	4.73	0.76	0.05	0.05	0.05	0.05
Standard deviation	26.98	23.35	29.82	29.83	29.81	29.82
Coefficient of variation	0.81	1.63	2.39	2.39	2.39	2.39

Related work

- IBM Cell/B.E. OpenMP compiler
  - Enable OpenMP model
  - Optimization of scalar code execution
  - autoSMIDization
  - Overlap data transfer with computation
- IBM Roadrunner
  - AMD Opteron processors are host elements and the Cell/BE. Blades are accelerator elements
- Accelerated Library Framework
  - Based on work queues
  - Supports (SPMD) programming style (Single program running on all accelerator elements at one time
  - Provides interface to easily partition data across a set of parallel processors

Related work

- Sequoia
  - Decompose of a program into tasks (like CellSs)
  - Tasks can call themselves recursively
  - The leaf task implementation

- RapidMind
  - Provides a set of data types, control flow macros, reduction operations, and common functions that allow the runtime library to capture a representation of the SPE code
  - Data types express SIMD easily
  - The runtime extracts parallelism from those operations by vetcorizing the code and by splitting array and vector calculations on different SPEs

Conclusions

- The objective is to offer a simple and flexible programming model for parallel and heterogeneous architectures
- Using CellSs, Cell Applications can be written as sequential programs
- Scheduling is based on a task-dependency graph of annotated functions
- Scheduling is locality aware
- CellSs is different from OpenMP as the latter requires the programmer to explicitly indicates what is parallel and what is not.
- OpenMP is more appropriate for applications with parallelism at the loop level, while CellSs fits better for parallelism at the function level