Cell Superscalar (CellSs) User's Manual

Version 2.2
Barcelona Supercomputing Center
May 2009



Contents

1 Introduction **1**

2 Installation **1**
2.1 Compilation requirements 1
2.2 Compilation 2
2.3 Runtime requirements 3
2.4 User environment 3

3 Programming with CellSs **4**
3.1 C Programming 4
3.1.1 Task selection 4
3.1.2 Specifying a task 4
3.1.3 Scheduling a task 6
3.1.4 Waiting on data 6
3.1.5 Mixed SPU and PPU code 7
3.2 Fortran Programming 8
3.2.1 Task selection 8
3.2.2 Specifying a task 9
3.2.3 Waiting on data 10
3.2.4 Fortran compiler restrictions 11

4 Compiling **12**
4.1 Usage 13
4.2 Examples 14

5 Setting the environment and executing **15**
5.1 Setting the number of SPUs and executing 15

6 Programming examples **16**
6.1 Matrix mutlitply 16

7 CellSs internals **18**

8 Advanced features **20**
8.1 Using paraver 20
8.2 Configuration file 22

9 CellSs SPU memory functionality **23**
9.1 Dynamic memory allocation 23
9.2 DMA accesses 23
9.3 *Strided* Memory Access 25

References **27**

List of Figures

1 CellSs runtime behavior 19
2 One dimensional memory access pattern. 25
3 Two dimensional memory access pattern. 26

1 Introduction

The Cell Broadband Engine (Cell/BE) is an heterogeneous multi-core architecture with nine cores. The first generation of the Cell BE includes a 64-bit multi-threaded PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB). The PPE has two levels of on-chip cache and also supports IBMs VMX to accelerate multimedia applications by using VMX SIMD units.

This document is the user manual of the Cell Superscalar (CellSs) framework, which is based on a source-to-source compiler and a runtime library. The programming model allows programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different components of the Cell/BE (PPE and SPEs) by means of an automatic parallelization at execution time. The requirements we place on the programmer are that the application is composed of coarse grain functions (for example, by applying blocking) and that these functions do not have collateral effects (only local variables and parameters are accessed). These functions are identified by annotations (somehow similar to the OpenMP ones), and the runtime will try to parallelize the execution of the annotated functions (also called tasks).

The source-to-source compiler separates the annotated functions from the main code and the library provides a manager program to be run in the SPEs that is able to call the annotated code. However, an annotation before a function does not indicate that this is a parallel region (as it does in OpenMP). To be able to exploit the parallelism, the CellSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule for execution independent nodes to different SPEs at the same time. All data transfers required for the computations in the SPEs are automatically performed by the runtime. Techniques imported from the computer architecture area like the data dependency analysis, data renaming and data locality exploitation are applied to increase the performance of the application.

While OpenMP explicitly specifies what is parallel and what is not, with CellSs what is specified are functions whose invocations could be run in parallel, depending on the data dependencies. The runtime will find the data dependencies and will determine, based on them, which functions can be run in parallel with others and which not. Therefore, CellSs provides programmers with a more flexible programming model with an adaptive parallelism level depending on the application input data and the number of available cores.

2 Installation

Cell Superscalar is distributed in source code form and must be compiled and installed before using it. The runtime library source code is distributed under the LGPL license and the rest of the code is distributed under the GPL license. It can be downloaded from the CellSs web page at http://www.bsc.es/cellsuperscalar .

2.1 Compilation requirements

The CellSs compilation process requires the following system components:

- GCC 4.1 or later.
- GNU make
- SPE Runtime Management Library, version 2.0 or higher.
- IBM XL Fortran multicore acceleration for Linux on System p. V11.1[1]. This is only necessary when CellSs Fortran support is required.

Additionally, if you change the source code you may require:

- automake
- autoconf $\geq$ 2.60
- libtool
- rofi-bison[2]
- GNU flex

2.2 Compilation

To compile and install CellSs please follow the following steps:

1. Decompress the source tarball.
   ```
   tar -xvzf CellSS-2.2.tar.gz
   ```
2. Enter into the source directory.
   ```
   cd CellSS-2.2
   ```
3. If necessary, check that you have set the *PATH* and *LD_LIBRARY_PATH* environment variables to point to the CBE SDK installation.
4. Run the configure script, specifying the installation directory as the prefix argument. The configure script also accepts the following optional parameters:
   - `--with-cellsdk=prefix` Specifies the CBE SDK installation path.

   More information can be obtained by running `./configure --help`.
   ```
   ./configure --prefix=/opt/CellSS
   ```
   There are also some environment variables that affect the configuration behaviour.
   - PPU C compiler may be specified with the `PPUCC` variable.
   - PPU C compiler flags may be given with the `PPUCFLAGS` variable.

[1] See http://www.alphaworks.ibm.com/tech/cellfortran
[2] Available at http://www.bsc.es/plantillaH.php?cat_id=351

- PPU C++ compiler may be specified with the `PPUCXX` variable.
- PPU C++ compiler flags may be given with the `PPUCXXFLAGS` variable.
- PPU Fortran compiler may be specified with the `PPUFC` variable.
- PPU Fortran compiler flags may be given with `PPUFCFLAGS`.

Note that currently our libraries only support 32 bits. If you choose to change any of the PPU compilers, then you must also pass the proper flag to force a 32 bits target. One simple way to do it is to force it into the command. For instance:

```
export PPUCC="/opt/sdk3.0/bin/ppu-gcc -m32"
```

- SPU C compiler may be specified with the `SPUCC` variable.
- SPU C compiler flags may be given with the `SPUCFLAGS` variable.
- SPU C++ compiler may be specified with the `SPUCXX` variable.
- SPU C++ compiler flags may be given with the `SPUCXXFLAGS` variable.
- SPU Fortran compiler may be specified with the `SPUFC` variable.
- SPU Fortran compiler flags may be given with `SPUFCFLAGS`.

5. Run make.

```
make
```

6. Run make install.

```
make install
```

2.3 Runtime requirements

The CellSs runtime requires the following system components:

- CBE SDK 3.0 or newer

2.4 User environment

If the CBE SDK resides on a non standard directory, then the user must set the *LD_LIBRARY_PATH* and *PATH* accordingly.

If CellSs has not been installed into a system directory, then the user must set the following environment variables:

1. The *PATH* environment variable must contain the bin subdirectory of the installation.

```
export PATH=$PATH:/opt/CellSS/bin
```

2. The *LD_LIBRARY_PATH* environment variable must contain the lib subdirectory from the installation.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/CellSS/lib
```

3 Programming with CellSs

CellSs applications are based on the parallelization at task level of sequential applications. The tasks (functions or subroutines) selected by the programmer will be executed in the SPE processors. Furthermore, the runtime detects when tasks are data independent between them and is able to schedule the simultaneous execution of several of them on different SPEs. Since, the SPE cannot access the main memory, the data required for the computation in the SPE is transferred by DMA. All the above mentioned actions (data dependency analysis, scheduling and data transfer) are performed transparently to the programmer. However, to take benefit of this automation, the computations to be executed in the Cell BE should be of certain granularity (about $50\mu s$). A limitation on the tasks is that they can only access their parameters and local variables. In case global variables are accessed the compilation will fail.

3.1 C Programming

The C version of CellSs borrows the syntax from OpenMP in the way that the code is annotated using special preprocessor directives. Therefore, the same general syntax rules apply; that is, directives are one line long but they can span into multiple lines by escaping the line ending.

3.1.1 Task selection

In CellSs, it is a responsibility of the application programmer to select tasks of certain granularity. For example, blocking is a technique that can be applied to increase the granularity of the tasks in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
void block_addmultiply (double C[BS][BS], double A[BS][BS],
                        double B[BS][BS])
{
    int i, j, k;

    for (i=0; i < BS; i++)
        for (j=0; j < BS; j++)
            for (k=0; k < BS; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

3.1.2 Specifying a task

A task is conceived in the form of a procedure, i.e. a function without return value. Then, a procedure is "converted" into a CellSs task by providing a simple annotation before its declaration or definition:

```
#pragma css task [input(<input parameters>)]optional \
                 [inout(<inout parameters>)]optional \
```

```
                    [output(<output parameters>)]optional \
                    [highpriority]optional
<function declaration or definition>
```

Where each clause serves the following purposes:

- `input` clause lists parameters whose input value will be read.
- `inout` clause lists parameters that will be read and written by the task.
- `output` clause lists parameters that will be written to.
- `highpriority` clause specifies that the task will be scheduled for execution earlier than tasks without this clause.

The parameters listed in the `input`, `inout` and `output` clauses are separated by commas. Only the parameter name and dimension(s) need to be specified, not the type. Although the dimensions must be omitted if present in the parameter declaration.

Remember that, when invoking a task, the issued parameters must must be aligned on a 4-byte boundary. Not fulfilling this requirement may produce DMA transfer errors or unexpected situations at the SPE.

Examples

In this example, the "factorial" task has a single input parameter "n" and a single output parameter "result".

```
#pragma css task input(n) output(result)
void factorial (unsigned int n, unsigned int *result)
{
    *result = 1;
    for (; n > 1; n--)
        *result = *result * n;
}
```

The next example, has two input vectors "left", of size "leftSize", and "right", of size "rightSize"; and a single output "result" of size "leftSize+rightSize".

```
#pragma css task input(leftSize, rightSize) \
                 input(left[leftSize], right[rightSize]) \
                 output(result[leftSize+rightSize])
void merge (float *left,  unsigned int leftSize,
            float *right, unsigned int rightSize, float *result)
{
    ...
}
```

The next example shows another feature. In this case, with the keyword `highpriority` the user is giving hints to the scheduler: the jacobi tasks will be, when data dependencies allow it, executed before the ones that are not marked as high-priority.

```
#pragma css task input(lefthalo[32], tophalo[32], righthalo[32], \
                       bottomhalo[32]) inout(A[32][32]) highpriority
void jacobi (float *lefthalo,  float *tophalo,
             float *righthalo, float *bottomhalo, float *A)
{
    ...
}
```

3.1.3 Scheduling a task

Once all the tasks have been specified, the next step is to use them. The way to do it is as simple as it gets: just call the annotated function normally. However, there still exists a small requirement, in order to have the tasks scheduled by the CellSs runtime, the annotated functions must be invoked in a block surrounded by these two directives:

```
#pragma css start
#pragma css finish
```

These two directives can only be used once in a program, i.e. it is not possible to annotate a `start` directive after[3] a `finish` directive has been annotated. They are also mandatory and must enclose all annotated function invocations.

Beware of the fact that the compiler will not detect these issues; the runtime will complain in some cases but may also incur into unexpected behaviour. Section 3.1.4 provides some simple examples.

3.1.4 Waiting on data

When code outside the tasks needs to handle data manipulated also by code inside the tasks, the automatic dependency tracking performed by the runtime is not enough to ensure correct read and write order. To solve this, CellSs offers some synchronization directives.

As in OpenMP, there is a barrier directive:

```
#pragma css barrier
```

This forces the main thread to wait for the completion of all generated tasks so far. However, this kind of synchronization is too coarse grained and in many cases can be counter productive. To achieve a finer grained control over data readiness, the `wait on` directive is also available:

[3] *After* in the code execution path.

```
#pragma css wait on(<list of variables>)
```

In this case, the main thread waits (or starts running tasks) until all the values of the listed variables have been committed. Like in other clauses[4], multiple variable names are separated by commas.

The data unit to be waited on should be consistent with the data unit of the task. For example, if the task is operating on the full range of an array, we cannot wait on a single element `arr[i]` but on its base address `arr`.

Examples

The next example shows how a `wait on` directive can be used:

```
#pragma css task inout(data[size]) input(size)
void bubblesort (float *data, unsigned int size)
{
    ...
}

void main ()
{
    ...
    #pragma css start

    bubblesort(data, size);

    #pragma css wait on(data)

    for (unsigned int i = 0; i < size; i++)
        printf("%f ", data[i]);

    #pragma css finish
}
```

In this particular case, a barrier could have served for the same purpose since there is just one output variable.

3.1.5 Mixed SPU and PPU code

The CellSs programming model allows by design to mix SPU and PPU code in the same source file. Tasks are always compiled for the SPU architecture. Functions reachable by tasks are also on the SPU side. All other functions are on the PPU side by default, including those reachable by them. These heuristics can lead to some functions appearing on both sides.

[4] Clauses that belong to the `task` directive.

However, the architecture selection heuristics are only applied individually to each source file. This could lead to some functions being compiled for the wrong architecture and producing linking errors. For instance, a function in one source file could be called by a task in another source file. Unless that function is also called by a task from the same source file, it will not be compiled for the SPU side.

To solve this problem, the architecture selection heuristics can be skipped by specifying explicitly the target architecture. Function definitions and declarations can have their target architecture(s) specified by using the following construct:

```
#pragma css target [spu]optional [ppu]optional
<function declaration or definition>
```

Functions can have either target or both simultaneously.

3.2 Fortran Programming

As in C, the Fortran version of CellSs also is based on the syntax of OpenMP for Fortran-95. This version of CellSs only supports free form code and needs some Fortran-95 standard features.

3.2.1 Task selection

In CellSs it is responsibility of the application programmer to select tasks of a certain granularity. For example, blocking is a technique that can be applied to increase such granularity in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
subroutine block_addmultiply(C, A, B, BS)
   implicit none
   integer, intent(in) :: BS
   real, intent(in) :: A(BS,BS), B(BS,BS)
   real, intent(inout) :: C(BS,BS)
   integer :: i, j, k

   do i=1, BS
     do j=1, BS
       do k=1, BS
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
       enddo
     enddo
   enddo
end subroutine
```

3.2.2 Specifying a task

A task is conceived in the form of a subroutine. The main difference with C CellSs annotations is that, in Fortran, the language provides the means to specify the direction of the arguments in a procedure. Moreover, while arrays in C can be passed as pointers, Fortran does not encourage that practice. In this sense, annotations in Fortran are simpler than in C. The annotations have the form of a Fortran-95 comment followed by a "$" and the framework sentinel keyword (`CSS` in this case). This is the same syntax that OpenMP uses in Fortran-95.

In Fortran, each subprogram calling tasks must know the interface for those tasks. For this purpose, the programmer must specify the task interface in the caller subprograms and also write some CellSs annotations to let the compiler know that there is a task. The following requirements must be satisfied by all Fotran tasks in CellSs:

- The task interface must specify the parameter directions of all parameters. That is, by using `INTENT (`*<direction>*`)`; where *<direction>* is one of: `IN`, `INOUT` or `OUT`.
- Provide an explicit shape for all array parameters in the task (caller subprogram).
- Provide a `!$CSS TASK` annotation for the caller subprogram with the task interface.
- Provide a `!$CSS TASK` annotation for the task subroutine.

The following example shows how a subprogram calling a CellSs task looks in Fortran. Note that it is not necessary to specify the parameter directions in the task subroutine, they are only necessary in the interface.

```
subroutine example()
   ...
   interface
     !$CSS TASK
     subroutine block_add_multiply(C, A, B, BS)
       ! This is the task interface. Specifies the size and
       ! direction of the parameters.
       implicit none
       integer, intent(in) :: BS
       real, intent(in) :: A(BS,BS), B(BS,BS)
       real, intent(inout) :: C(BS,BS)
     end subroutine
   end interface
   ...
   !$CSS START
   ...
   call block_add_multiply(C, A, B, BLOCK_SIZE)
   ...
   !$CSS FINISH
   ...
```

```
end subroutine

!$CSS TASK
subroutine block_add_multiply(C, A, B, BS)
   ! Here goes the body of the task (the block multiply_add
   ! in this case)
   ...
end subroutine
```

It is also necessary, as the example shows, to call the tasks between `START` and `FINISH` annotation directives. These are executable statements that must be after the declarations, in the executable part of the subprogram. `START` and `FINISH` statements must only be executed once in the application.

Example

The following example shows part of a CellSs application using another feature. The `HIGHPRIORITY` clause is used to indicate that one task is high priority and must be executed before non-high priority tasks as soon as its data dependencies allow.

```
interface
   !$CSS TASK HIGHPRIORITY
   subroutine jacobi(lefthalo, tophalo, righthalo, bottomhalo, A)
     real, intent(in), dimension(32) :: lefthalo, tophalo, &
                                        righthalo, bottomhalo
     real, intent(inout) :: A(32,32)
   end subroutine
end interface
```

3.2.3 Waiting on data

CellSs provides two different ways of waiting on data. These features are useful when the user wants to read the results of some tasks in the main thread. By default tasks will write their results back to main memory after its execution. Nevertheless, the main thread does not know when these results are ready. With `BARRIER` and `WAIT ON`, the main program stops its execution until the data is available.

`BARRIER` is the most conservative option. When the main thread reaches a barrier, waits until all tasks have finished and have written their results back to main memory. The syntax is simple:

```
   ...
   do i=1, N
     call task_a(C(i),B)
   enddo
   ...
   !$CSS BARRIER
```

```
   print *, C(1)
   ...
```

The other way is to specify exactly which variables we want the program to wait to be available before the execution goes on.

```
!$CSS WAIT ON(<list of variables>)
```

Where the list of variables is a comma separated list of variable names whose values must have been computed before continuing the execution.

Example

```
!$CSS TASK
subroutine bubblesort (data, size)
    integer, intent(in) :: size
    real, intent(inout) :: data(size)
    ...
end subroutine

program main
    ...
    interface
      !$CSS TASK
      subroutine bubblesort (data, size)
        integer, intent(in) :: size
        real, intent(inout) :: data(size)
      end subroutine
    end interface
    ...
    call bubblesort(data, size);

    !$CSS WAIT ON(data)

    do i=1, size
      print *, data(i)
    enddo
end
```

3.2.4 Fortran compiler restrictions

This is the first release of CellSs with a Fortran compiler and it has some limitations. Some will disappear in the future. They consist of compiler specific and non-standard features. Also deprecated forms in the Fortran-95 standard are not supported and are not planned to be included in future releases.

- Case sensitiveness: The CellSs Fortran compiler is case insensitive. However, task names must be written in lowercase.
- It is not allowed to mix generic interfaces with tasks.
- Internal subprograms cannot be tasks.
- Use of modules within tasks has not been tested in this version.
- Optional and named parameters are not allowed in tasks.
- Some non-standard common extensions like Value parameter passing are not supported or have not been tested yet. In further releases of CellSs we expect to support a subset of the most common extensions.
- Only explicit shape arrays and scalars are supported as task parameters.
- The `MULTOP` parameter is not supported.
- Tasks cannot have an `ENTRY` statement.
- Array subscripts cannot be used as task parameters.
- `PARAMETER` arrays cannot be used as task parameters.

4 Compiling

The CellSs compiler infastructure is composed of a C99 source-to-source compiler, a Fortran-95 source-to-source compiler and a common driver.

The driver is called `cellss-cc` and depending on each source filename suffix invokes transparently the C compiler or the Fortran-95 compiler. C files must have the "`.c`" suffix. Fortran files can have either the "`.f`", "`.for`", "`.f77`", "`.f90`", or "`.f95`" suffix, they can also be in uppercase in case preprocessing is desired.

The `cellss-cc` driver behaves similarly to a native compiler. It can compile individual files one at a time, several ones, link several objects into an executable or perform all operations in a single step.

The compilation process consists in processing the CellSs pragmas, transforming the code according to those, compiling both for the PPU and SPU with the corresponding compilers (`ppu-c99` and `spu-c99` for C programs and, in the case of Fortran programs, with `ppuxlf95` and `spuxlf`), and packing both objects and additional information required for linking into a single object..

The linking process consists in unpacking the object files, generating additional code required for the SPU part, compiling it, linking all SPU objects together, embedding the SPU executable into a PPU object (`ppu32-embedspu`), generating additional code required for the PPU part , compiling it, and finally linking all PPU objects together with the CellSs runtime to generate the final executable.

4.1 Usage

The `cellss-cc` compiler has been designed to mimic the options and behaviour of common C compilers. However, it uses two other compilers internally that may require different sets of compilation options.

To cope with this distinction, there are general options and target specific options. While the general options are applied to PPU code and SPU code, the target specific options allow to specify options to pass to the PPU compiler and the SPU compiler independently.

The list of supported options is the following:

```
> cellss-cc -help

Usage: cellss-cc <options and sources>

Options:
  -D<macro>           Defines 'macro' with value '1' in the
                      preprocessor.
  -D<macro>=<value>   Defines 'macro' with value 'value' in the
                      preprocessor.
  -g                  Enables debugging.
  -h|--help           Shows usage help.
  -I<directory>       Adds 'directory' the list of preprocessor
                      search paths.
  -k|--keep           Keeps intermediate source and object files.
  -l<library>         Links with the specified library.
  -L<directory>       Adds 'directory' the list of library search
                      paths.
  -O<level>           Enables optimization level 'level'.
  -o <filename>       Sets the name of the output file.
  -c                  Specifies that the code must only be compiled
                      (and not linked).
  -t|--tracing        Enables run time tracing.
  -v|--verbose        Enables verbose operation.

PPU specific options:
  -WPPUp,<options>    Passes the comma separated list of options to
                      the PPU C preprocessor.
  -WPPUc,<options>    Passes the comma separated list of options to
                      the PPU C compiler.
  -WPPUf,<options>    Passes the comma separated list of options to
                      the PPU Fortran compiler.
  -WPPUl,<options>    Passes the comma separated list of options to
                      the PPU linker.

SPU specific options:
```

```
  -WSPUp,<options>    Passes the comma separated list of options to
                      the SPU C preprocessor.
  -WSPUc,<options>    Passes the comma separated list of options to
                      the SPU C compiler.
  -WSPUf,<options>    Passes the comma separated list of options to
                      the SPU Fortran compiler.
  -WSPUl,<options>    Passes the comma separated list of options to
                      the SPU linker.
```

4.2 Examples

Contrary to previous versions of the compiler, now it is possible to generate binaries from multiple source files like any regular C or Fortran95 compiler. Therefore, it is possible to compile multiple source files directly into a single binary:

```
> cellss-cc -O3 *.c -o my_binary
```

Although handy, you may also use the traditional compilation methodology:

```
> cellss-cc -O3 -c code1.c
> cellss-cc -O3 -c code2.c
> cellss-cc -O3 -c code3.f90
> cellss-cc -O3 code1.o code2.o code3.o -o my_binary
```

This capability allows to easily adapting makefiles by just changing the C compiler, the Fortran compiler and the linker to point to `cellss-cc`. For instance:

```
CC = cellss-cc
LD = cellss-cc
CFLAGS = -O2 -g

SOURCES = code1.c code2.c code3.c
BINARY = my_binary

$(BINARY): $(SOURCES)
```

Combining the `-c` and `-o` options makes possible to generate objects with arbitrary filenames. However, changing the suffix to other than `.o` is not recommended since, in some cases, the compiler driver relies on them to work properly.

As already mentioned, the same binary serves as a Fortran95 compiler:

```
> cellss-cc -O3 matmul.f90 -o matmul
```

If there are no compilation errors, the executable file "matmul" (optimized) is created and can be called from the command line ("`> ./matmul ...`").

In some cases, it is desirable to use specific optimization options not included in the `-O`, `-O1`, `-O2`, or `-O3` set. This is possible by using the `-WSPUc` and/or `-WPPUc` flags (depending on the kind of target that requires the optimization):

```
> cellss-cc -O2 -WSPUc,-funroll-loops,-ftree-vectorize \
            -WSPUc,-ftree-vectorizer-verbose=3 matmul.c -o matmul
```

In the previous example, the native options are passed directly to the native compiler (for example `spu-c99`), to perform automatic vectorization of the code to be run in the SPUs.

Option `-k`, or `--keep`, will not delete the intermediate files (files generated by the preprocessor, object files, ...).

```
> cellss-cc -k cholesky.c -o cholesky
```

Finally, option `-t` enables executable instrumentation to generate a runtime trace to be analyzed later with the appropriate tool:

```
> cellss-cc -O2 -t matmul.c -o matmul
```

When executing "matmul", a trace file of the execution of the application will be generated. See section 8.1 for further information on trace analysis.

5 Setting the environment and executing

Depending on the path chosen for installation (see section 2), the *LD_LIBRARY_PATH* environment variable may need to be set appropriately or the application will not be able to run.

If CellSs was configured with `--prefix=/foo/bar/CellSS`, then *LD_LIBRARY_PATH* should contain the path */foo/bar/CellSS/lib*. If the framework is installed in a system location such as */usr*, setting the loader path is not necessary.

5.1 Setting the number of SPUs and executing

Before executing a CellSs application, the number of SPU processors to be used in the execution have to be defined. The default value is 8, but it can be set to a different number with the *CSS_NUM_SPUS* environment variable, for example:

```
> export CSS_NUM_SPUS=6
```

CellSs applications are started from the command line in the same way as any other application. For example, for the compilation examples of section 4.2, the applications can be started as follow:

```
> ./matmul <pars>
> ./cholesky <pars>
```

6 Programming examples

This section presents a programming example for the block matrix multiplication. The code is not complete, but you can find the complete and working code under *programs/matmul* in the installation directory. More examples are also provided in this directory.

6.1 Matrix mutlitply

This example presents a CellSs code for a block matrix multiply. The block contains $BS \times BS$ floats.

```
#pragma css task input(A, B) inout(C)
static void block_addmultiply (float C[BS][BS], float A[BS][BS],
                               float B[BS][BS])
{
    int i, j, k;

    for (i = 0; i < BS; i++)
        for (j = 0; j < BS; j++)
            for (k = 0; k < BS; k++)
                C[i][j] += A[i][k] * B[k][j];
}

int main(int argc, char **argv)
{
    int i, j, k;

    initialize(argc, argv, A, B, C);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                block_addmultiply(C[i][j], A[i][k], B[k][j]);
    ...
}
```

The main code will run in the Cell PPE while the `block_addmultiply` calls will be executed in the SPE processors. It is important to note that the sequential code (including the annotations) can be compiled and run in a sequential processor. This is very useful for debugging the algorithms.

However, the code is not vectorized, and if a compiler that does not vectorize the code is used, it is not going to be very efficient. The programmer can pass to the corresponding compiler the compilation flags that automatically vectorize the SPU (see section 4.2). Another option will be to manually provide a vectorized code as the one that follows:

```
#define BS 64
#define BSIZE_V BS/4
#pragma css task input(A, B) inout(C)
void block_addmultiply (float C[BS][BS], float A[BS][BS],
                        float B[BS][BS])
{
    vector float *Bv = (vector float*) B;
    vector float *Cv = (vector float*) C;
    vector float elem;
    int i, j, k;

    for (i = 0; i < BS; i++)
        for (j = 0; j < BS; j++) {
            elem = spu_splats (A[i * BS + j]);
            for (k = 0; k < BSIZE_V; k++)
                Cv[i*BSIZE_V+k] = spu_madd(elem, Bv[j * BSIZE_V + k],
                                           Cv[i * BSIZE_V + k]);
        }
}
```

This code can even be improved by unrolling the inner loop. Even more, the code can be improved if the data is prefetched in advance, as the next version of the sample code does:

```
#define BS 64
#define BSIZE_V BS/4
#pragma css task input(A, B) inout(C)
void matmul (float A[BSIZE][BSIZE], float B[BSIZE][BSIZE],
             float C[BSIZE][BSIZE])
{
    vector float *Bv = (vector float *) B;
    vector float *Cv = (vector float *) C;

    vector float elem;
    int i, j;
    int i_size;
    int j_size;
    vector float tempB0, tempB1, tempB2, tempB3;

    i_size = 0;
    for (i = 0; i < BSIZE; i++) {
        j_size = 0;
```

```
        for (j = 0; j < BSIZE; j++) {
            elem = spu_splats(A[i][j]);
            tempB0 = Bv[j_size+0];
            tempB1 = Bv[j_size+1];
            tempB2 = Bv[j_size+2];
            Cv[i_size+0] = spu_madd(elem, tempB0, Cv[i_size+0]);
            tempB3 = Bv[j_size+3];
            Cv[i_size+1] = spu_madd(elem, tempB1, Cv[i_size+1]);
            tempB0 =  Bv[j_size+4];
            Cv[i_size+2] = spu_madd(elem, tempB2, Cv[i_size+2]);
            tempB1 =  Bv[j_size+5];
            Cv[i_size+3] = spu_madd(elem, tempB3, Cv[i_size+3]);
            tempB2 =  Bv[j_size+6];
            Cv[i_size+4] = spu_madd(elem, tempB0, Cv[i_size+4]);
            tempB3 =  Bv[j_size+7];
            Cv[i_size+5] = spu_madd(elem, tempB1, Cv[i_size+5]);
            tempB0 =  Bv[j_size+8];
            Cv[i_size+6] = spu_madd(elem, tempB2, Cv[i_size+6]);
            tempB1 =  Bv[j_size+9]
            Cv[i_size+7] = spu_madd(elem, tempB3, Cv[i_size+7]);
            tempB2 =  Bv[j_size+10];
            Cv[i_size+8] = spu_madd(elem, tempB0, Cv[i_size+8]);
            tempB3 =  Bv[j_size+11];
            Cv[i_size+9] = spu_madd(elem, tempB1, Cv[i_size+9]);
            tempB0 =  Bv[j_size+12];
            Cv[i_size+10] = spu_madd(elem, tempB2, Cv[i_size+10]);
            tempB1 =  Bv[j_size+13];
            Cv[i_size+11] = spu_madd(elem, tempB3, Cv[i_size+11]);
            tempB2 =  Bv[j_size+14];
            Cv[i_size+12] = spu_madd(elem, tempB0, Cv[i_size+12]);
            tempB3 =  Bv[j_size+15];
            Cv[i_size+13] = spu_madd(elem, tempB1, Cv[i_size+13]);
            Cv[i_size+14] = spu_madd(elem, tempB2, Cv[i_size+14]);
            Cv[i_size+15] = spu_madd(elem, tempB3, Cv[i_size+15]);
            j_size += BSIZE_V;
        }
        i_size += BSIZE_V;
    }
}
```

7 CellSs internals

When compiling a CellSs application with `cellss-cc`, the resulting object files are linked with the CellSs runtime library. Then, when the application is started, the CellSs runtime is automatically invoked. The CellSs runtime is decoupled in two parts: one runs in the PPU and the other in each of

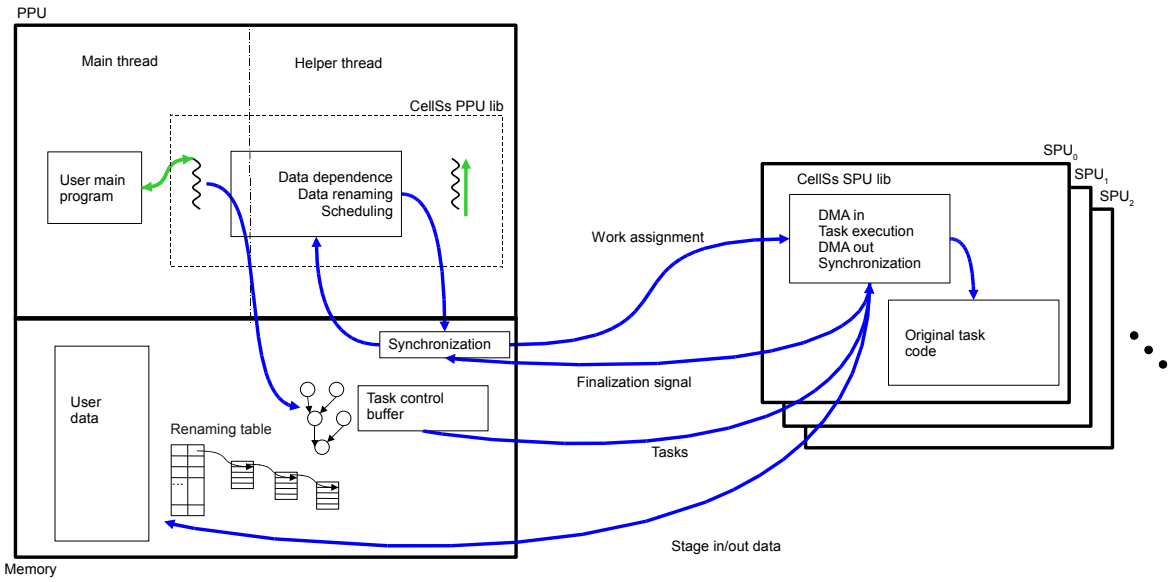


Figure 1: CellSs runtime behavior

the SPUs. In the PPU, we will differentiate between the master thread and the helper thread.

The most important change in the original user code is that the CellSs compiler replaces calls to tasks with calls to the `css_addTask` function. At runtime, these calls will be responsible for the intended behavior of the application in the Cell BE processor. At each call to `css_addTask`, the master thread will do the following actions:

- Add node that represents the called task in a task graph.
- Analyze data dependencies of the new task with other previously called tasks.
- Parameter renaming: similarly to register renaming, a technique from the superscalar processor area, we do renaming of the output parameters. For every function call that has a parameter that will be written, instead of writing to the original parameter location, a new memory location will be used, that is, a new instance of that parameter will be created and it will replace the original one, becoming a renaming of the original parameter location. This allows to execute that function call independently from any previous function call that would write or read that parameter. This technique allows to effectively remove some data dependencies by using additional storage, and thus improving the chances to extract more parallelism.

The helper thread is the one that decides when a task should be executed and also monitors the execution of the tasks in the SPUs.

Given a task graph, the helper thread schedules tasks for execution in the SPUs. This scheduling follows some guidelines:

- A task can be scheduled if its predecessor tasks in the graph have finished their execution.

- To reduce the overhead of the DMA, groups of tasks are submitted to the same SPU.
- Data locality is exploited by keeping task outputs in the SPU local memory and scheduling tasks that reuse this data to the same SPU.

The helper thread synchronizes and communicates with the SPUs using a specific area of the PPU main memory for each SPU. The helper thread indicates the length of the group of tasks to be executed and information related to the input and output data of the tasks.

The SPUs execute a loop waiting for tasks to be executed. Whenever a group of tasks is submitted for execution, the SPU starts the DMA of the input data, processes the tasks and writes back the results to the PPU memory. The SPU synchronizes with the PPU to indicate end of the group of tasks using a specific area of the PPU main memory.

8 Advanced features

8.1 Using paraver

To understand the behavior and performance of the applications, the user can generate Paraver [3] tracefiles of their CellSs applications.

If the `-t/-tracing` flag is enabled at compilation time, the application will generate a Paraver tracefile of the execution. The default name for the tracefile is gss-trace-id.prv. The name can be changed by setting the environment variable *CSS_TRACE_FILENAME*. For example, if it is set as follows:

```
> export CSS_TRACE_FILENAME=tracefile
```

After the execution, the files: tracefile-0001.row, tracefile-0001.prv and tracefile-0001.pcf are generated. All these files are required by the Paraver tool.

The traces generated by CellSs can be visualized and analyzed with Paraver. Paraver [3] is distributed independently of CellSs.

Several configuration files to visualise and analyse CellSs tracefiles are provided in the CellSs distribution in the directory *<install_dir>/share/cellss/paraver_cfgs/*. The following table summarizes what is shown by each configuration file.

Configuration file	Feature shown
2dh_inbw.cfg	Histogram of the bandwidth achieved by individual DMA IN transfers. Zero on the left, 10GB/s on the right. Darker colour means more times a transfer at such bandwidth occurred.
2dh_inbytes.cfg	Histogram of bytes read by the stage in DMA transfers.

Configuration file	Feature shown
2dh_outbw.cfg	Histogram of the bandwidth achieved by individual DMA OUT transfers. Zero on the left, 10GB/s on the right. Darker colour means more times a transfer at such bandwidth occurred.
2dh_outbytes.cfg	Histogram of bytes writen by the stage out DMA transfers.
3dh_duration_phase.cfg	Histogram of duration for each of the runtime phases.
3dh_duration_tasks.cfg	Histogram of duration of SPU tasks. One plane per task (Fixed Value Selector). Left column: 0 microseconds. Right column: 300 us. Darker colour means higher number of instances of that duration.
DMA_bw.cfg	DMA (in + out) bandwidth per SPU.
DMA_bytes.cfg	Bytes being DMAed (in + out) by each SPU.
execution_phases.cfg	Profile of percentage of time spent by each thread (main, helper and SPUs) at each of the major phases in the runt time library (i.e. generating tasks, scheduling, DMA, task execution, ...).
flushing.cfg	Intervals (dark blue) where each SPU is flushing its local trace buffer to main memory. For the main and helper threads the flushing is actually to disk. Overhead in this case is thus significant as this stalls the respective engine (task generation or submission).
general.cfg	Mix of timelines.
stage_in_out_phase.cfg	Identification of DMA in (grey) and out phases (green).
task.cfg	Outlined function being executed by each SPU.
task_distance_histogram.cfg	Histogram of task distance between dependent tasks.
task_number.cfg	Number (in order of task generation) of task being executed by each SPU. Ligth green for the initial tasks in program order, blue for the last tasks in program order. Intermixed green an blue indicate out of order execution.
Task_profile.cfg	Time (microseconds) each SPU spent executing the different tasks. Change statistic to: • #burst: number of tasks of each type by SPU. • Average burst time: Average duration of each task type.
task_repetitions.cfg	Shows which SPU executed each task and the number of times that the task was executed.
Total_DMA_bw.cfg	Total DMA (in+out) bandwidth to Memory.

8.2 Configuration file

With the objective of tuning the behaviour of the CellSs runtime, a configuration file where some variables are set is introduced. However, we do not recommend to play with them unless the user considers that it is required to improve the performance of her/his applications. The current set of variables is the following (values between parenthesis denote the default value):

- scheduler.initial_tasks (128): this number specifies how many tasks must be ready before scheduling for the first time.
- scheduler.min_tasks (16): defines minimum number of ready tasks before they are scheduled (no more tasks are scheduled while this number is not reached).
- scheduler.max_strand_size (8): defines the maximum number of tasks that are simultaneously scheduled to an SPE.
- spe.stack_size (4096 bytes): defines the size of the area of the local store dedicated for the runtime stack.
- spe.softcache_size (variable): defines the size of the area of the local store where the worker stores and caches task arguments. By default, the size equals the local store space between the $.BSS$ section of the binary, and the maximum top of the stack as defined by spe.stack_size.
- task_graph.task_count_high_mark (1000): defines the maximum number of non-executed tasks that the graph will hold.
- task_graph.task_count_low_mark (900): whevever the task graph reaches the number of tasks defined in the previous variable, the task graph generation is suspended until the number of non-executed tasks goes below this amount.
- renaming.memory_high_mark (∞): defines the maximum amount of memory used for renaming in bytes.
- renaming.memory_low_mark (1): whenever the renaming memory usage reaches the size specified in the previous variable, the task graph generation is suspended until the renaming memory usage goes below the number of bytes specified in this variable.

This variables are set in a plain text file, with the following syntax:

```
scheduler.min_tasks             = 32
scheduler.initial_tasks         = 128
scheduler.max_strand_size       = 8
task_graph.task_count_high_mark = 2000
task_graph.task_count_low_mark  = 1500
renaming.memory_high_mark       = 134217728
renaming.memory_low_mark        = 104857600
```

The file where the variables are set is indicated by setting the *CSS_CONFIG_FILE* environment variable. For example, if the file "file.cfg" contains the above variable settings, the following command can be used:

```
> export CSS_CONFIG_FILE=file.cfg
```

Some examples of configuration files for the execution of CellSs applications can be found at location *<install_dir>/share/docs/cellss/examples/*.

9 CellSs SPU memory functionality

9.1 Dynamic memory allocation

Local Storage (LS) space in each SPU is limited, so CellSs tries to control as much of it as possible. In particular, the layout of the libraries does not permit the use of the heap[5]. CellSs has its own rudimentary memory allocator, where heap space and code managing can be subsumed; as a result, more LS space becomes available. In return, *all* task code must use the dynamic memory allocation interface offered by CellSs.

This interface differs only syntactically from the familiar `libc` counterpart. If CellSs has been installed in `prefix`, then this header file can be found in *prefix/worker/*. Allocated space must be freed by the end of the task. Failure to do so will cause it to be lost for the remainder of the execution. Memory obtained through this interface will always be 16-byte aligned. Large local variables should be allocated using this interface, instead of being pushed on the stack because, by default, CellSs reserves 4 kilobytes of stack space.

```
#include "css_malloc.h"

void *css_malloc (unsigned int size);
void  css_free   (void *chunk);
```

9.2 DMA accesses

Although CellSs handles all data transfers for the parameters in the tasks interface, in some cases the programmer may want to be able to do explicit data transfers from main memory. From a CellSs task, the user can access main memory via the following set of DMA routines. All accesses are asynchronous, and the locations in main memory should be 16-byte aligned. For transfers of 1, 2, 4, 8 bytes, or multiples of 16 bytes up to 16 kilobytes (or 16384 bytes), the interface offers the following functions:

[5]Which might involve serious problems if system libraries or user libraries include calls to malloc or free.

```
#include "css_dma_red.h"

void css_get_a (void *ls, uint32_t ea, unsigned int dma_size,
                tagid_t tag);
void css_put_a (void *ls, uint32_t ea, unsigned int dma_size,
                tagid_t tag);
```

Where each argument stands for:

- `ls` is a pointer to a 16-byte aligned user-allocated buffer in LS.
- `ea` is the pointer to main memory where the buffer resides that holds the object to be transfered to `ls`, or where the object `ls` points to will be transfered.
- `dma_size` is the size of the object in bytes.
- `tag` is the identifier of the DMA transfer.

A DMA transfer or a group of DMA transfers are identified by a *tag*. Any number of transfers can be grouped together by using the same tag. After starting an asynchronous transfer, its completion can be assured via the DMA tag. A tag corresponds to a number between 0 and 31, and the user is free to choose from this range. However, in order to avoid grouping together unrelated DMA transfers, the user should request a DMA tag:

```
tagid_t css_tag (void);
```

Range	Use
0 – 7	short-circuit + dummy
8 – 15	stage in
16 – 23	stage out + alternating
24 – 31	reserved

Table 2: Use of DMA tags in the worker library.

The worker library itself divides the group of available tags according to the scheme in table 2. These functions start the asynchronous DMA transfers. To check for their completion, the user performs a call to:

```
void css_sync (tagid_t tag)
```

DMA transfers that do not comply to the criteria outlined above (transfers of size diferent than 1, 2, 4, 8, and multiples of 16 bytes), should use the following interface instead:

```
void css_get (void *ls, unsigned int address, unsigned int size,
              tagid_t tag);
void css_put (void *ls, unsigned int address, unsigned int size,
              tagid_t tag);
```

9.3 *Strided* Memory Access

CellSs offers an interface to scatter/gather memory access patterns for one, two and three dimensional arrays. The only requirement is that all parts are aligned on a 16-byte boundary. Each of the functions in this interface returns and accepts a parameter called `c_list`. If a preceding call to a function from this interface accessed main memory according to a certain pattern, this pattern can be reused by passing it the `c_list` created by that call.

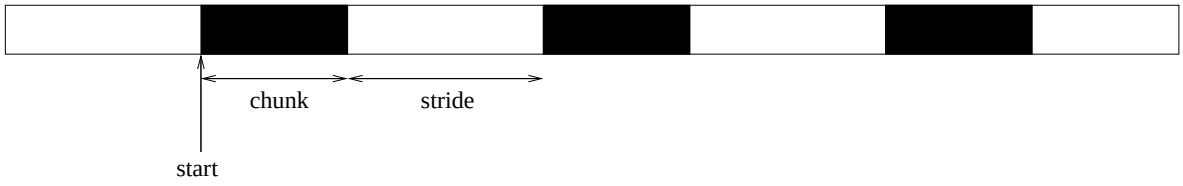


Figure 2: One dimensional memory access pattern.

CellSs implements its scatter/gather functionality via DMA lists. The reuse of `c_list` objects enables the worker library to in turn reuse those DMA lists, instead of recreating them from scratch. `esize` is the size of a single argument from the matrix in bytes. `size` is the total number of objects you want to scatter/gather. `ls` is a pointer to a 16-byte aligned buffer in LS that has been previously allocated by the user. `start` is a pointer to main memory, indicating the address of the very first element of the matrix that will be collected (gather), or the address of the location where the first element will be put (scatter). `ls` is a pointer to LS, which is the beginning of the buffer that will contain the objects to be gathered, or the beginning of the buffer that contains the objects to be scattered. `start` and `ls` must be 16-byte aligned.

For the one dimension case, the situation is depicted in figure 2, and the interface is:

```
#include "css_stride_red.h"

dmal_h_t *css_gather_1d  (void *ls, unsigned int start, int chunk,
                          int stride, size_t size, size_t e_size,
                          dmal_h_t *c_list);
dmal_h_t *css_scather_1d (void *ls, unsigned int start, int chunk,
                          int stride, size_t size, size_t e_size,
                          dmal_h_t *c_list);
```

For the two dimensions case, the situation is depicted in figure 3, and the interface is:

```
#include "css_stride_red.h"
```

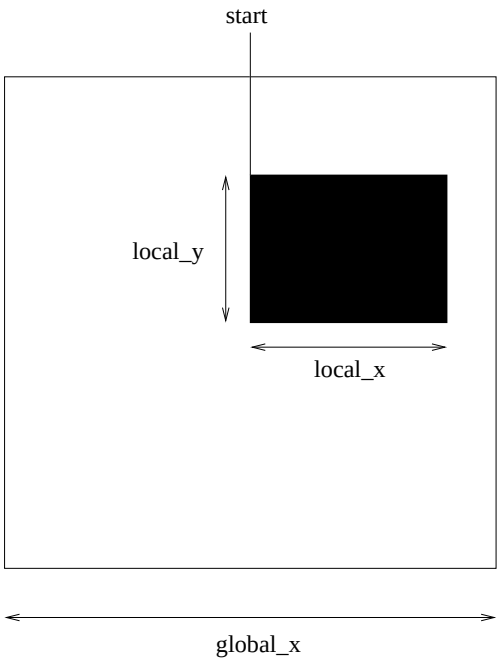


Figure 3: Two dimensional memory access pattern.

```
dmal_h_t *css_gather_2d  (void *ls, unsigned int start, int local_x,
                          int local_y, int global_x, size_t e_size,
                          dmal_h_t *c_list);
dmal_h_t *css_scather_2d (void *ls, unsigned int start, int local_x,
                          int local_y, int global_x, size_t e_size,
                          dmal_h_t *c_list);
```

The three dimensions case is an extension of the two dimensions case, adding an extra dimension:

```
#include "css_stride_red.h"

dmal_h_t *css_gather_3d  (void *ls, unsigned int start, int local_x,
                          int local_y, int local_z, int global_x,
                          int global_z, size_t e_size,
                          dmal_h_t *c_list);
dmal_h_t *css_scather_3d (void *ls, unsigned int start, int local_x,
                          int local_y, int local_z, int global_x,
                          int global_z, size_t e_size,
                          dmal_h_t *c_list);
```

The above DMA operations are asynchronous. After invoking these functions and starting the DMAs, the user should wait for their completion before accessing the data they transfer. Each transfer has associated with it a DMA tag, that can be retrieved through the `tag` field of the `dmal_h_t` object returned by the initial invocation. For example, the following code extract illustrates how to use `css_gather_1d` in a CellSs task:

```
#pragma css task input(A[16*16],A_p)
void matmul (float *A, unsigned int A_p)
{
#ifdef SPU_CODE
     dmal_h_t *entry = css_gather_1d(A, A_p, 4, 16, 128, sizeof(float), NULL);
     short tag = entry->tag;

     css_sync(tag);
#endif
}
```

Remark that the declaration of `A` as a task argument ensures that there will be a 16×16 byte buffer available, and that the actual direction of `A` in main memory gets passed via `A_p`. An alternative is to use `css_malloc` and `css_free` to manage buffers inside the task.

References

[1] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[2] Barcelona Supercomputing Center. Cell Superscalar website. http://www.bsc.es/cellsuperscalar.

[3] CEPBA/UPC. Paraver website. http://www.bsc.es/paraver.

[4] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. CellSs: Programming the Cell/B.E. made easier. *IBM Journal of R&D*, 51(5), August 2007.