NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

INTERDEPARTMENTAL PROGRAM OF POSTGRADUATE STUDIES IN MICROELECTRONICS

MASTER THESIS

Implementation of Linear Algebra Kernels in Heterogeneous Systems using StarPU

Stavroula G. Zouzoula

ATHENS

APRIL 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΗ ΜΙΚΡΟΗΛΕΚΤΡΟΝΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Πυρήνων Γραμμικής Άλγεβρας σε Ετερογενή Συστήματα χρησιμοποιώντας την StarPU

Σταυρούλα Γ. Ζούζουλα

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2021

MASTER THESIS

Implementation of Linear Algebra Kernels in Heterogeneous Systems using StarPU

Stavroula G. Zouzoula
R.N.: MM306

SUPERVISOR: Dimitrios Soudris, Professor National Technical University of Athens

THREE-MEMBER EXAMINATION COMMITTEE

Dimitrios Soudris, Professor National Technical University of Athens
Antonis Pashalis, Professor National and Kapodistrian University of Athens
Lazaros Papadopoulos, Doctor National Technical University of Athens

April 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Πυρήνων Γραμμικής Άλγεβρας σε Ετερογενή Συστήματα χρησιμοποιώντας την StarPU

Σταυρούλα Γ. Ζούζουλα
Α.Μ.: MM306

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Δημήτριος Σούντρης, Καθηγητής Εθνικό Μετσόβιο Πολυτεχνείο

ΤΡΙΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Δημήτριος Σούντρης, Καθηγητής Εθνικό Μετσόβιο Πολυτεχνείο
Αντώνης Πασχάλης, Καθηγητής Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Λάζαρος Παπαδόπουλος, Διδάκτωρ Εθνικό Μετσόβιο Πολυτεχνείο

Απρίλιος 2021

ABSTRACT

Manycore and heterogeneous computing systems led application programmers to turn to parallelism. A lot of parallel models have been proposed, with the most promising being task-based models. Many task-based runtime systems have been developed in the effort to better exploit parallelism and achieve better performance.

In this master thesis, StarPU's runtime system is presented. StarPU was applied to some Polybench benchmarks that were executed for different input sizes in an heterogeneous node. From the obtained results, we reached conclusions on StarPU's scheduling decisions.

SUBJECT AREA: Task-based Runtime Systems

KEYWORDS: Parallel programming, task-based programming models, StarPU

ΠΕΡΙΛΗΨΗ

Τα πολυπήρυνα και εταιρογενή υπολογιστικά συστήματα οδήγησαν τους προγραμματιστές εφαρμογών να στραφούν στον παραλληλισμό. Έχουν προταθεί πολλά παράλληλα μοντέλα, με το πιο πολλά υποσχόμενο να είναι τα μοντέλα βασισμένα σε διεργασίες. Πολλά συστήματα εκτέλεσης χρόνου βασισμένα σε διεργασίες έχουν αναπτυχθεί στην προσπάθεια να εκμεταλλευτούν καλύτερα τον παραλληλισμό και να επιτύχουν καλύτερες επιδόσεις.

Σε αυτή τη μεταπτυχική διπλωματική, παρουσιάζεται το σύστημα εκτέλεσης χρόνου της StarPU. Η StarPU εφαρμόστηκε σε μερικές εφαρμογές απ' το Polybench όπου εκτελέστηκαν για διάφορα μεγέθη εισόδου σε έναν ετερογενή κόμβο. Από τα ληφθέντα αποτελέσματα, καταλήξαμε σε συμπεράσματα σχετικά με τις αποφάσεις προγραμματισμού της StarPU.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συστήματα Χρόνου Εκτέλεσης Βασισμένα σε Διεργασίες

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Παράλληλος προγραμματισμός, προγραμματιστικά μοντέλα βασισμένα σε διεργασίες, StarPU

ACKNOWLEDGEMENTS

Upon completion of this master thesis, I would like to thank Professor Dimitrios Soudris for his help, support and the trust he showed in me. I would also like to warmly thank Dr. Lazaros Papadopoulos for his guidance, immense help and the excellent cooperation we had. Finally, I would like to thank my family for always being by my side and supporting me throughout my studies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CODES

1. INTRODUCTION

In the past years, processor designers tried to achieve better performance by increasing clock frequency. During this process they had to face some challenges like energy consumption and heat dissipation. The solution to these challenges was the switch to multicore processors. However, applications were developed following the sequential execution model, thus they could not take advantage of the multicore architectures in order to increase their performance. This issue led application developers to parallel programming.

There are two main methods for applications' parallelization, auto parallelization and parallel programming [14]. In auto parallelization, applications developed using the sequential programming model are automatically parallelized. This is achieved through instruction level parallelism (ILP) or parallel compilers. On the other hand, in parallel programming, applications are specifically developed to exploit parallelism by using a parallel programming model.

There are various parallel programming models. Some parallel programming models are parallel random access machine (PRAM), data parallel models and task-based parallel models [15]. In the PRAM model, there is a set of processors connected to a shared memory, fed by a global clock. All processors may access shared memory simultaneously. In data parallel models the same scalar computation is applied on different data. All data computations must be independent to each other and thus, they can be performed in parallel. In task-based parallel models, applications can break into tasks, where each task solves a part of the problem. Tasks may communicate with other tasks while executing, or send results to other tasks with their termination. This master thesis focuses on task-based parallel models.

A crucial role in task-based parallel models plays the underlying runtime system (RTS) which encompasses the implementation mechanism. It finds concurrent accesses to the data arguments and schedules the tasks to available processing units. There are many RTSs that leverage the task-based programming model. In this work, we focus on StarPU (see Chapter 3). StarPU was applied to a set of applications (mostly linear algebra kernels) to demonstrate the flexibility of execution of these kernels in heterogeneous systems, based on StarPU's scheduling decisions. The results of this work were published in the 2nd International Workshop on Parallel Optimization using/for Multi- and Many-core High Performance Computing (POMCO 2020) [27].

1.1 Motivation

In the high-performance computing (HPC) domain, platforms are becoming more and more complex as they integrate many cores and accelerators (e.g GPUs, FPGAs). The heterogeneity of the platforms arose some burdens to the application programmers, such as efficient scheduling, managing the data movements and communication between the

distributed memory machines. Runtime systems address these challenges by integrating sophisticated algorithms, to enable load balancing, efficient data movement and optimized scheduling considering criteria such as performance and energy consumption.

In this work, we evaluate the widely used StarPU task-based RTS. Firstly, we want to observe how task-based RTSs respond to kernels used primarily in HPC applications. Secondly, we would like to ascertain that the advanced scheduling features of the task-based RTSs lead to efficient scheduling of the tasks. Finally, we want to estimate how easy it is for a user unfamiliar with RTSs to use them.

1.2 Structure of the document

Chapter 2 discusses task-based models, what leads to their wide acceptance and their benefits. There are references to some models, a comparison of their features and some indicative results of past work based on those models.

Chapter 3 presents the StarPU runtime system. It describes basic aspects of the system and how they are used in order to build a StarPU code. It, also, provides a simple example of the original and the StarPU-ized version of a program.

Chapter 4 contains the evaluation of StarPU. There is a brief description of the applications used in the thesis and their results. There is also a reference in the training and programming effort.

Chapter 5 lists the conclusions and future directions of the present study.

2. TASK-BASED RUNTIME SYSTEMS

2.1 Task-based parallel programming

When multicore architectures emerged, in order to achieve higher performance, programmers turned to parallelism. However, accelerators like GPUs, gained popularity and are being adopted in computing due to the benefits they offer. That lead to the heterogeneity of the architectures. Parallelism alone, was not able to cope with the challenges that many-core heterogeneous architectures brought with them. A prominent solution is task-based parallel programming.

In task-based parallel programming, a task dependence graph (TDG) is being built. The TDG is a directed acyclic graph (DAG), where nodes represent tasks and directed edges represent dependencies between tasks. Later, an RTS is responsible for assigning tasks to resources, such as CPU cores and accelerators, and managing data transfers. That leaves the programmer to focus on the implementation of efficient computational kernels.

This paradigm has been acceptable and widely used because of the many advantages it offers. Some of them are:

- Higher level description: The programmer has to resolve only logical dependencies between tasks.
- Automatic task scheduling: The task scheduler is mapping tasks to logical threads[1] exploiting higher level information like task dependencies, task types, etc, so as to avoid interference from other threads.
- Better load balancing: The task scheduler distributes work evenly to the threads.
- Tasks are light weight: It is faster to start and shutdown tasks compared to logical threads, as the latter have their own copies of resources.

2.2 Task-based models

Over the past decades, a lot of task-based models have been proposed, but not all of them have still active developers. In this section we are going to discuss some of them.

OpenMP

In Open Multi-Processing (OpenMP[2]) [3], for executing the program in parallel, the application developer has to explicitly specify the actions to be taken by the compiler and RTS.

[1] The scheduler tries to have one logical thread per physical thread

[2] Current release at the time of writing this master thesis is OpenMP 5.1. Main webpage: `https://www.openmp.org`

This is being accomplished with the use of directives that extend the C, C++ and Fortran languages. The directives are of the form **#pragma omp *construct*** *[clause[[,]clause]...]* for C/C++ and **!$omp *construct*** *[clause[[,]clause]...]* for Fortran. The constructs **task** and **taskwait** were added for support of explicit tasks and task synchronization respectively, in version 3.0 of OpenMP execution model. In version 4.0, task dependencies were added to the model with the clause **depend**. Up to the current version, the model is being updated to support more task-based functions, like non-blocking join operations in a TDG with the **nowait** clause.

StarSs

Star Superscalar (StarSs) [21] exploits task-level parallelism by annotating the code with compiler directives. The form of the directives is **#pragma css *construct*** *[clause-list]*. The tasks are being specified with the construct **task**. The construct **target** and the clause **device** are being used to specify that the implementation of the task is for a specific architecture, in heterogeneous systems[3]. The programmer specifies the *tasks* inside the application and the direction of the parameters. With this information, the RTS builds a TDG that composes the application.

OmpSs-2

OmpSs-2[4] [2] is an extension of OmpSs [12], which is a combination of the OpenMP standard and StarSs. The main objective is to extend OpenMP so as to better support asynchronous data flow parallelism and heterogeneity. OmpSs-2 uses directives to execute an application in a parallel architecture. Their format is **#pragma oss *directive-name*** *[clause[[,]clause]...]* for C/C++ and ***sentinel directive-name*** *[clause[[,]clause]...]* for Fortran, where *sentinel* depends on Fonrtran's fixed/free form and it could be one of **!$oss**, **c$oss** or ***$oss**. The programmer specifies the tasks with the construct **task**. With the clause **depend** or with just the type of the dependence as the name of the clause, task scheduling restrictions are being declared. For synchronization of the tasks, the construct **taskwait** or **taskwait on***(list-of-variables)* can be used, with the first resulting to wait for deep completion of all descendant tasks and the second one resulting to wait for a subset of descendant tasks that had a dependence on variables from the variable list.

HPX

High Performance ParallelX (HPX)[5] [1][13] is an open source, C++, task-based RTS. It is used for programming parallel and distributed applications, but it does not support hetero-

[3] It supports only heterogeneous systems with shared memory

[4] Main webpage: `https://pm.bsc.es/ompss-2`

[5] Main webpage: `http://stellar.cct.lsu.edu/projects/hpx/`, Repository: `https://github.com/STEllAR-GROUP/hpx/`

geneous architectures. Programming in HPX is like C++ programming under the namespace **hpx**. The most important terms of this RTS are **async** and **future**. The first one is used for asynchronous execution of the tasks, with return type *future<T>*. The latter one represents the result from an operation that is being retrieved with proper synchronization, if it belongs to different threads.

Cilk

Cilk is a programming language that extends C and C++, to allow data and task parallelism. The first version of the language was Cilk [7], that was a parallel programming extension to the C language. A thread was being defined similarly to a C function, **thread *T*** *(args) {statements}* and it could spawn a child thread with the instruction **spawn *T*** *(args)*. Later on, Cilk++ [17] was developed to extend C++. That was achieved with three keywords, **cilk_spawn** that creates parallel work when it precedes a function call, **cilk_sync** that is a local barrier that provides explicite synchronization and **cilk_for** that allows the parallel operation of the loop iterations. Cilk Plus[6] [23] was the successor of Cilk++ which introduced vector parallelism. Currently, research into Cilk technology continuous with OpenCilk[7], which is an open-source implementation of Cilk, that incorporates several enhancements.

TBB

Threading Building Blocks (TBB)[8] [28] is a C++ library that provides a high level abstraction for parallelism. Initially, it was developed for manycore homogeneous with shared memory systems. However, work is being done towards heterogeneity, with the goal to create a coordination layer that will compose with existing models. It is worth mentioning **asynch_node** and **opencl_node** that are being used in heterogeneous devices. The programming is being done in terms of tasks instead of threads, under the namespace **tbb**. **parallel_invoke** is being used to create tasks that different worker threads can execute them in parallel. **parallel_for** divides the iterations range into chunks, which, with a body, consist the tasks that are scheduled to threads that execute the algorithm. For synchronization, *mutex classes* or *atomic classes* are being used.

[6] Cilk Plus was deprecated in the 2018 release of Intel Software Development Tools, according to Intel's Community anouncement in `https://community.intel.com/t5/Software-Archive/Intel-Cilk-Plus-is-being-deprecated/td-p/1127776`

[7] Main webpage: `https://cilk.mit.edu`

[8] Main webpage: `https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top.html`

2.3 Comparison of features

Table 2.1 compares the features of the task-based models mentioned in Section 2.2 and the task-based model described in Chapter 3. The information contained in the table is the implementation type of the model, the support of heterogeneous systems and memory architecture (shared/distributed), the graph structure, the scheduling method, the way tasks are being created and how they are synchronized.

Some notes about the information contained in the table.

- Programmers can choose from a range of implementation types, depending on what suits them best.
- Most of the models have turned to heterogeneity.
- It was a common practice to develop those models for shared memory systems, but there are also models that investigate the distributed memory models.
- There are models with various scheduling policies, such as *priority local*, *static* and *dynamic*.
- Most of the models synchronize the task implicitly, but also provide ways for explicit synchronization.

2.4 Indicative results

The evaluation of the efficiency of a model is usually being done by comparing it to another model, most commonly to OpenMP and PThreads. There are several studies that compare different task-based models in the literature such as [22][18][25].

Figure 2.1a represents the execution time of the STREM beanchmark when changing the array size, using 32 processors for OpenMP and SMPSs[9]. This figure was taken from [6], where in that work they tried to understand why the versions (static/dynamic for OpenMP and barriers/no barriers for StarSs) of the models scale with the number of processors and the role that memory locality plays to the scaling. They showed that SMPSs performs better than OpenMP, when they use the mechanism that inserts a new ready task, added by the SMPSs runtime, directly to the ready list of the thread that first touched the data that the task accesses.

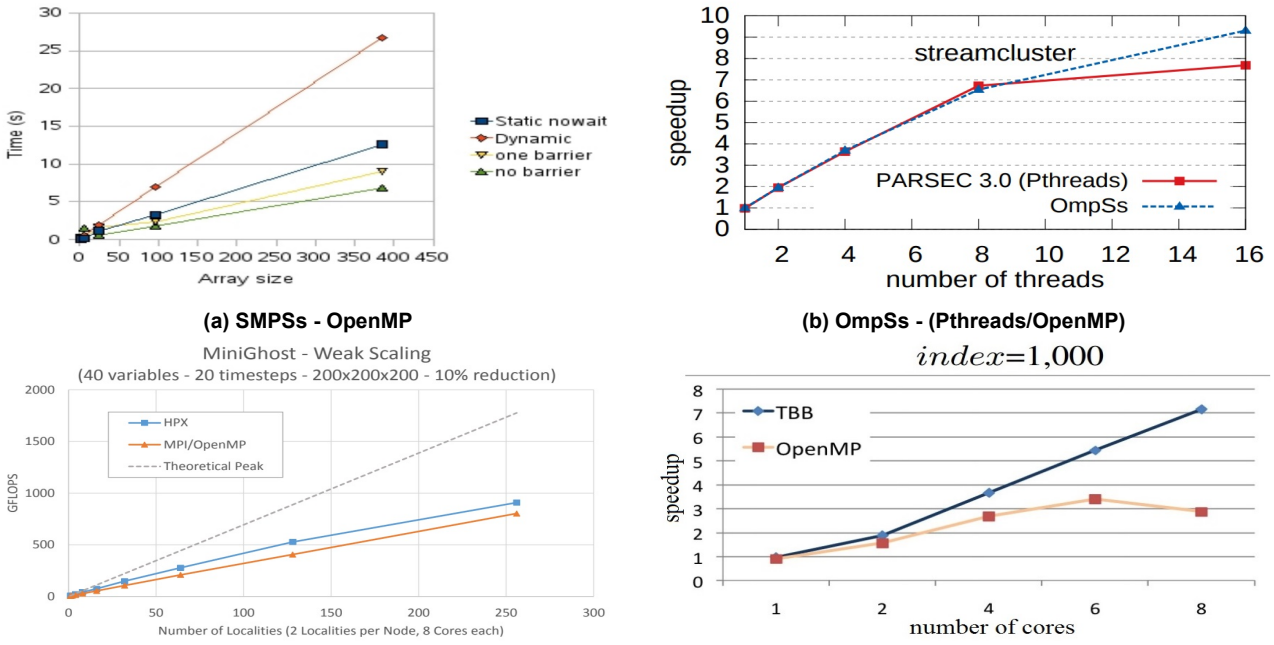
In [8] the scalability of the task-based versions, using OmpSs, of the PARSEC benchmarks, compared to the Pthread/OpenMP version was investigated. Figure 2.1b is a sample of their results. The conclusion of that work was that it is easier to use task-based models in a wide range of applications and there are cases where it can offer better scalability, but there are also applications that do not benefit from a task-based approach.

[9] SMPSs is one of StarSs's environments that share the same *pragma* syntax

Table 2.1: Feature comparison of task-based models

Features	OpenMP	StarSs	OmpSs	HPX	Cilk	TBB	StarPU
Imp. type	Extension	Extension	Extension	Library	Language	Library	Extension
Heterogeneity	Yes	Yes	Yes	Yes	No	Yes*	Yes
Memory	Shared	Shared	Shared	Distributed	Shared	Shared	Distributed
Graph Structure	DAG	DAG	DAG	DAG	Tree	Tree	DAG
Scheduling	Various	Various	Various	Various	Work Stealing	Work Stealing	Various
Task Creation	#pragma omp task	#pragma css task	#pragma oss task	hpx_call**	cilk_spawn	tbb_call**	starpu_task_create() starpu_task_insert()
Task Synch.	implicit/ explicit	implicit	implicit/ explicit	implicit/ explicit	explicit	implicit/ explicit	explicit

* There is no full support of heterogeneous systems at the time of writing this thesis
** Thread creation



(a) SMPSs - OpenMP

(b) OmpSs - (Pthreads/OpenMP)

(c) HPX - MPI/OpenMP

(d) TBB - OpenMP

Figure 2.1: Indicative results of StarSs, OmpSs, HPX and TBB models

Figure 2.1c depicts the performance results of the miniGhost application in HPX and MPI/OpenMP for distributed runs, in the work presented in [13]. In that work is shown that HPX performs better for shared, as well as distributed runs due to the effectiveness of the ”futurization” of the control flow in the first case and to the fine grained constraints that result in efficient overlapping of communication and computation.

The work conducted in [10] evaluates the suitability of TBB in parallelization of wavefront problems. Figure 2.1d shows the speedups of their experiments for input matrix size 1000×1000 and index 1000, for different number of cores. Their conclusions were that the scalability of the code is affected by the granularity of the task and that TBB outperforms OpenMP, except when the task has coarse granularity, where both implementation appear to show similar behavior.

3. THE STARPU RUNTIME SYSTEM

StarPU[1] [5][4] is a unified RTS that can address various architectures, like CPUs, GPUs, etc., and hence the name, from the symbolism *PU. It supports a task-based programming model, where tasks with CPU and/or GPU implementations are being submitted to StarPU's RTS, which schedules the tasks on available resources.

StarPU's programming model is the Sequential Task Flow (STF) model [26], which allows programmers to submit tasks in the sequential flow of the program and the tasks are being submitted asynchronously. The RTS is responsible for detecting task parallelism through data dependencies (and thus creating a TDG) and task scheduling.

Any code using StarPU should include the header `<starpu.h>`. In addition, before any StarPU call StarPU's initialization method (`starpu_init(NULL)`[2]) should be called. At the end of the application, the termination method (`starpu_shutdown()`) should be called.

3.1 StarPU basics

The main structures of StarPU are **codelets** and **tasks**. A codelet represents a computational kernel that could be implemented on various architectures, like CPU, GPU and OpenCL devices. A task describes a codelet, the data that are going to be accessed and the access mode of the data (read, write, read/write).

3.1.1 Codelet

A codelet is a structure that describes a kernel. The field `.where` specifies the types of processing units that the codelet can be executed on, i.e the available implementations of the kernel. The values of the field can be `STARPU_CPU`, `STARPU_CUDA`, `STARPU_OPENCL` and `STARPU_NOWHERE` which indicates that no computation has to be done. We are going to focus only on CPU and CUDA implementations. When there are multiple implementations, they are separated with a vertical bar (`|`). Another field is the `.*_funcs`, where * can be `cpu`, `cuda` and `opencl`[3]. It is an array of function pointers to the implementations of the codelet. This field is ignored if the `.where` field is set. Along with `.cpu_funcs`, the field `.cpu_funcs_name` can be used. This field is an array of strings that contain the name of the CPU functions in `.cpu_funcs`. Two necessary fields inside the codelet are `.nbuffers` and `.modes`. The first one specifies the non constant arguments manipulated by the codelet, while the latter describes their access mode (`STARPU_R` for read-only, `STARPU_W` for write-only, `STARPU_RW` for read-write).

[1] Current release at the time of writing this master thesis is 1.3.7, Main webpage: `https://starpu.gitlabpages.inria.fr`, Repository: `https://gitlab.inria.fr/starpu/starpu/-/tree/master`

[2] The NULL argument specifies the use of the default configuration

[3] On the git repository of StarPU, there is also support for FPGAs (in the fpga branch), thus the `.fpga_funcs` can be used

```
static struct starpu_codelet codelet_name =
{
  .where = available_implemetations
  .*_funcs = { *_implementation_name },   // * could be cpu, cuda, opencl
  .cpu_funcs_name = { "name_of_cpu_function" },
  .nbuffers = number_of_arguments,
  .modes = { access_mode_of_the_data },
  ...
};
```

3.1.2 Data manipulation

In StarPU, programmers have to declare only the data that the task is going to access and/or modify and StarPU will take care of data transfers before any computation. Data are managed with handles. `starpu_data_handle_t` keeps track of the piece of data associated to it. The different pieces of data are declared by the functions `starpu_*_data_register`, where * can be void, variable, vector, matrix, block, etc. The arguments of the `starpu_*_data_register` functions differ depending on the data interface. However, the first argument is always a pointer to the designated data. Below is described the way of using those functions.

starpu_void_data_register
Registers a void interface with no data being associated to it. It can be used as a synchronization mechanism.

```
starpu_void_data_handle(&handle_name);
```

starpu_variable_data_register
Registers a variable (i.e a given-size byte element), typically a scalar. The arguments of the function are the handle that represents the variable, the node number where the data originally reside[4], a pointer to the variable and the size of the variable.

```
starpu_variable_data_handle(&handle_name, home_node, (uintptr_t)&variable,
    sizeof(variable));
```

starpu_vector_data_register
Registers a vector (i.e a fixed number of elements of a given size). The arguments of the function are the handle that represents the vector, the node number where the data originally reside, a pointer to the vector, the number of vector elements, and the element size.

```
starpu_vector_data_handle(&handle_name, home_node, (uintptr_t)vector, NX,
    sizeof(vector[0]));
```

[4]The values of `home_node` are usually `STARPU_MAIN_RAM` for data in the main memory and -1 for not allocated data. In the latter case, memory is allocated automatically when it is used for the first time in write-only mode

starpu_matrix_data_register
Registers a 2D matrix with a potential padding. The arguments of the function are the handle that represents the matrix, the node number where the data originally reside, a pointer to the matrix, the number of elements between rows, the number of elements in x-axis, the number of elements in y-axis and the element size.

```
starpu_matrix_data_handle(&handle_name, home_node, (uintptr_t)matrix, LD, NX,
    NY, sizeof(element_type));
```

starpu_block_data_register
Registers a 3D matrix with a potential padding on Y and Z dimensions. The arguments of the function are the handle that represents the 3D matrix, the node number where the data originally reside, a pointer to the matrix, the number of elements between rows, the number of elements between z planes, the number of elements in x-axis, the number of elements in y-axis, the number of elements in z-axis and the element size.

```
starpu_block_data_handle(&handle_name, home_node, (uintptr_t)block, LDY, LDZ,
    NX, NY, NZ, sizeof(element_type));
```

3.1.3 Task

A StarPU task represents a codelet's execution on some data handles. In order to execute a codelet, the programmer has either to submit or insert a task.

Submit task
The first step of submitting a task, is to allocate the task structure by calling the function `starpu_task_create()`. The next step is to configure the task structure with the information about the codelet and the piece of data that the codelet should operate on. This is done by setting the fields of the task structure. Some of the fields are the following:

- `cl`: Pointer to the corresponding codelet that describes where the kernel should be executed.
- `handles[i]`: Specifies the handles associated to the different pieces of data accessed by the task. The number of task handles is specified in the `.nbuffers` field of the codelet.
- `cl_arg`: Pointer to a buffer with parameters for the kernel described by the codelet.
- `cl_arg_size`: Allocates on the driver a buffer of size `cl_arg_size`, starting at address `cl_arg`.
- `where`: Specifies where the task is allowed to be executed. When set, the codelet's field `.where` is ignored.
- `synchronous`: Explicit synchronization. If set (i.e $integer = 1$), then task submit is a blocking operation and returns only when the task has been executed.

The last step is to submit the task by calling the function `starpu_task_submit(task_name)`. This function returns 0 in case of success and `-ENODEV` when there is no worker able to execute the task. Some examples of task submission failure are trying to submit a task on a specific device, but the device is not available (e.g task with only CUDA implementation and no GPU available) and trying to submit an OpenCL task without having disabled CUDA first.

```
struct starpu_task *task_name = starpu_task_create();
task_name->cl = &codelet_name;
task_name->handles[i] = handle_name_i;
task_name->cl_arg = &parameter;
task_name->cl_arg_size = sizeof(parameter);
task->where = STARPU_**;          // ** can be CPU, CUDA, OPENCL, NOWHERE
task->synchronous = integer;
starpu_task_submit(task_name);
```

Insert task

`starpu_task_insert()` is a wrapper function that creates and submits a task. Below are the main arguments of the function. The final argument of `starpu_task_insert()` must be 0.

- `codelet_name`: Specifies the codelet that corresponds to the task.
- `STARPU_TASK_SYNCHRONOUS`: Explicit synchronization. It is followed by an integer. If the integer is 1 then the task is synchronous, if it is 0 then the task remains asynchronous.
- `access mode`: It specifies the access mode of the different pieces of data accessed by the task. It must be followed by the handle associated to the corresponding data. The access modes and the number of task handles are specified in the `.mode` and `.nbuffers` fields of the codelet respectively.
- `STARPU_VALUE`: Specifies a constant value. It must be followed by a pointer to the constant and the size of it.

```
starpu_task_insert(&codelet_name,
                STARPU_TASK_SYNCHRONOUS, integer,
                access_mode_i, handle_name_i,
                STARPU_VALUE, parameter, sizeof(parameter),
                0);
```

StarPU tasks are asynchronous (except if explicitly stated as shown before), which creates the need for some kind of synchronization. A way of achieving synchronization is the use of the `starpu_task_wait_for_all` function which blocks until all the submitted tasks are terminated. Another way is by using the `starpu_data_unregister` function that takes a handle as an argument. `starpu_data_unregister` will implicitly wait for all the tasks scheduled to work on the specific piece of data specified by the handle to terminate.

3.1.4 Codelet implementations

When a codelet is about to execute, the corresponding device will call the function declared by the `*_implementation_name`, as described in Section 3.1.1. The function's prototype must be:

```
1 void *_implementation_name(void *descr[], void *_args)
```

The first argument is an array that contains pointers to the different interfaces and gives a description of the input and output buffers passed in the `handles` array. The second argument is a pointer to optional arguments of the codelet. StarPU has defined macros in order to access the different data interfaces. Following is the description of some of those macros for CPU and CUDA devices only.

Variable data interface

- `STARPU_VARIABLE_GET_PTR(descr[i])`: Returns a pointer to the variable designated by `descr[i]`, a pointer to a variable interface.
- `STARPU_VARIABLE_GET_ELEMSIZE(descr[i])`: Returns the size of the designated variable.

Vector data interface

- `STARPU_VECTOR_GET_PTR(descr[i])`: Returns a pointer to the array designated by `descr[i]`, a pointer to a vector interface.
- `STARPU_VECTOR_GET_NX(descr[i])`: Returns the number of elements registered into the designated array.
- `STARPU_VECTOR_GET_ELEMSIZE(descr[i])`: Returns the size of the elements of the designated array.

Matrix data interface

- `STARPU_MATRIX_GET_PTR(descr[i])`: Returns a pointer to the matrix designated by `descr[i]`, a pointer to a matrix interface.
- `STARPU_MATRIX_GET_NX(descr[i])`: Returns the number of elements on the x dimension of the designated matrix.
- `STARPU_MATRIX_GET_NY(descr[i])`: Returns the number of elements on the y dimension of the designated matrix.
- `STARPU_MATRIX_GET_LD(descr[i])`: Returns the number of elements between each row of the designated matrix.
- `STARPU_MATRIX_GET_ELEMSIZE(descr[i])`: Returns the size of the elements of the designated matrix.

Block data interface

- `STARPU_BLOCK_GET_PTR(descr[i])`: Returns a pointer to the block (i.e 3D matrix) designated by `descr[i]`, a pointer to a block interface.
- `STARPU_BLOCK_GET_NX(descr[i])`: Returns the number of elements on the x dimension of the designated block.
- `STARPU_BLOCK_GET_NY(descr[i])`: Returns the number of elements on the y dimension of the designated block.
- `STARPU_BLOCK_GET_NZ(descr[i])`: Returns the number of elements on the z dimension of the designated block.
- `STARPU_BLOCK_GET_LDY(descr[i])`: Returns the number of elements between each row of the designated block.
- `STARPU_BLOCK_GET_LDZ(descr[i])`: Returns the number of elements between each z plane of the designated block.
- `STARPU_BLOCK_GET_ELEMSIZE(descr[i])`: Returns the size of the elements of the designated block.

The next lines show, in a generic way, how to use those macros in the implementation's function.

```
1 data_type *ptr1 = (data_type *)STARPU_**_GET_PTR(descr[i]);   // ** can be
    VARIABLE, VECTOR, MATRIX, etc
2 data_type name_i = STARPU_**_***(descr[i]);     // *** can be any of the
    options described (e.g GET_NX, GET_ELEMSIZE, etc)
```

3.2 Performance models

StarPU is able to estimate beforehand the duration of a task, by defining a performance model structure and addressing it in the codelet at the `.model` field. The performance model structure has two mandatory fields, `.type` and `.symbol` that specify the type and the name of the performance model respectively. Some performance model types are the following:

- *STARPU_HISTORY_BASED*: It is done per task size and is measured at runtime. It uses as an estimation the average execution time of previous executions on the various processing units.
- *STARPU_REGRESSION_BASED*: It works with different data input sizes, by applying regression over observed execution times. There should be at least 10% difference between the minimum and maximum input sizes. It is measured at runtime and is being refined by an $a * n^b$ regression form.

- *STARPU_NL_REGRESSION_BASED*: It is similar to STARPU_REGRESSION_-BASED, but it is refined by an $a * n^b + c$ regression form.

The definition of the performance model structure is:

```
static struct starpu_perfmodel model_name = {
    .type = performanc_model_type,
    .symbol = "model_name"
};
```

The programmer has to address the performance model inside the codelet in the field `.model`. He has, also, to initialize the model in the main code by calling the function `starpu_perfmodel_init`.

```
.model = &model_name    // address performance model in the codelet
starpu_perfmodel_init(&performance_model_name);   // initialize performance
   model in the main code
```

When using performance models, StarPU forces the calibration of the codes in order to avoid bad scheduling decisions. The calibration is done until 10 measurements have been made. The programmer can enable or disable code calibration by setting the environment variable `STARPU_CALIBRATE` (`export STARPU_CALIBRATE = 1 or 0`).

3.3 Scheduling policies

StarPU offers a number of different scheduling policies, that usually contain one or more queues that are associated with the different workers. Tasks are being submitted (push) to those queues or pulled (pop) for execution.

The available schedulers can be categorized according to if a performance model has been used or not. Some schedulers of each category are the following.

Non performance modelling

- *eager*: All workers draw tasks from a central task queue.
- *ws (work stealing)*: Each worker has its own queue from where it draws tasks. If one worker is idle, it steals tasks from the worker with the heaviest load.
- *lws (locality work stealing)*: This scheduler is similar to the *ws* with the differences being that when it's idle it steals tasks from the neighboring worker and it takes into account priorities. This is the default StarPU scheduler.

Performance modelling

- *dm (deque model)*: It schedules available tasks without taking into account priorities, where their termination time will be minimal.

- *dmda (deque model data aware)*: It is almost the same as *dm* with the difference being that it takes into account data transfer time.
- *peager (parallel eager)*: It is similar to *eager* scheduler with the support of parallel tasks.

The programmer can select a different scheduling policy either with the help of the environment variable `STARPU_SCHED` (`export STARPU_SCHED=scheduler`), or by setting the scheduler to the `sched_policy_name` field in the configuration structure `struct starpu_conf`. `starpu_conf_init` initializes the configuration structure with the default values. When the configuration structure is set, StarPU's initialization method should be called with the configuration structure as an argument (`starpu_init(&conf_name)`).

```
struct starpu_conf conf_name;
starpu_conf_init(&conf_name);
conf.sched_policy_name = "scheduling_policy";
conf.sched_policy_init = &scheduling_policy;
starpu_init(&conf_name);
```

3.4 A simple StarPU-ized example

In this Chapter, so far there was a presentation of some basic characteristics of StarPU. Now, let's apply this knowledge to a simple example, addition of two vectors. There are three vectors (`vec1, vec2, result`) dynamically allocated. `vec1` and `vec2` are initialized with random numbers between 0 and 10. The result of the vector addition is stored in the `result` vector. Listing 3.1 shows the C code of the example. Listing 3.2 and 3.3 show the StarPU-ized version of the example code, with the first having the CPU implementation of the kernel and the other containing the main code.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  int *vec1 = (int *)malloc(10*sizeof(int));
  int *vec2 = (int *)malloc(10*sizeof(int));
  int *result = (int *)malloc(10*sizeof(int));
  int i;

  /* initialize vectors */
  for (i = 0; i < 10; i++){
    vec1[i] = rand()%10;
    printf("%d ", vec1[i]);
  }

  printf("\n");

  for (i = 0; i < 10; i++){
    vec2[i] = rand()%10;
```

```
    printf("%d ", vec2[i]);
  }

  /* perform vector addition */
  for (i = 0; i < 10; i++)
    result[i] = vec1[i] + vec2[i];

  /* print result vector */
  printf ("\nAddition vector:\n");

  for (i = 0; i < 10; i++)
    printf ("%d ", result[i]);

  free(vec1);
  free(vec2);
  free(result);
}
```

Code 3.1: C program for vectors addition

```
#include <starpu.h>

void vecsum_cpu(void *descr[], void *_args)
{
  printf("cpu function\n");

  /* length of the vector */
  unsigned n = STARPU_VECTOR_GET_NX(descr[0]);

  /* local copy of the vector pointers */
  int *vec1 = (int *)STARPU_VECTOR_GET_PTR(descr[0]);
  int *vec2 = (int *)STARPU_VECTOR_GET_PTR(descr[1]);
  int *result = (int *)STARPU_VECTOR_GET_PTR(descr[2]);

  int i;

  /* perform vector addition */
  for (i = 0; i < n; i++)
    result[i] = vec1[i] + vec2[i];

}
```

Code 3.2: Vector addition kernel, CPU implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <starpu.h>

extern void vecsum_cpu(void *desc[], void *_args);

/* Define History-based performance model */
static struct starpu_perfmodel perf_model = {
  .type = STARPU_HISTORY_BASED,
  .symbol = "vecsum",
```

```
};

/* Define codelet */
static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { vecsum_cpu },
  .cpu_funcs_name = { "vecsum_cpu" },
  .nbuffers = 3,
  .modes = { STARPU_R, STARPU_R, STARPU_W },
  .model = &perf_model
};

int main ()
{
  int *vec1 = (int *)malloc(10*sizeof(int));
  int *vec2 = (int *)malloc(10*sizeof(int));
  int *result = (int *)malloc(10*sizeof(int));
  int i;

  /* initialize vectors */
  for (i = 0; i < 10; i++){
    vec1[i] = rand()%10;
    printf("%d ", vec1[i]);
  }

  printf("\n");

  for (i = 0; i < 10; i++){
    vec2[i] = rand()%10;
    printf("%d ", vec2[i]);
  }

  /* initialize StarPU with default configuration */
  int ret = starpu_init(NULL);
  if (ret == -ENODEV) goto enodev;

  /* initialize performance model */
  starpu_perfmodel_init(&perf_model);

  /* associate data with handles */
  starpu_data_handle_t vec1_handle, vec2_handle, res_handle;
  starpu_vector_data_register(&vec1_handle, STARPU_MAIN_RAM, (uintptr_t)vec1,
   10, sizeof(vec1[0]));
  starpu_vector_data_register(&vec2_handle, STARPU_MAIN_RAM, (uintptr_t)vec2,
   10, sizeof(vec2[0]));
  starpu_vector_data_register(&res_handle, STARPU_MAIN_RAM, (uintptr_t)result,
    10, sizeof(result[0]));

  /* insert task */
  starpu_task_insert( &cl,
            STARPU_R, vec1_handle,
            STARPU_R, vec2_handle,
```

```
61              STARPU_W, res_handle,
62              0);
63
64    /* stop monitoring the vectors */
65    starpu_data_unregister(vec1_handle);
66    starpu_data_unregister(vec2_handle);
67    starpu_data_unregister(res_handle);
68
69    /* terminate StarPU */
70    starpu_shutdown();
71
72    /* print result vector */
73    printf ("\nAddition vector:\n");
74    for (i = 0; i < 10; i++)
75      printf ("%d ", result[i]);
76
77    free(vec1);
78    free(vec2);
79    free(result);
80
81    enodev:
82      return 77;
83  }
```

Code 3.3: StarPU-ized vector addition

3.5 StarPU in EXA2PRO

Enhancing Programmability and boosting Performance Portability for Exascale Computing Systems (EXA2PRO)[5] [24] is a European Commission funded project[6]. The aim of the project is to implement a programming environment that will enable the efficient deployment of highly parallel algorithms of complex scientific problems in exascale (10^{18} operations per second) computing systems. EXA2PRO is integrating tools to address exascale challenges. One of those tools is StarPU. In the context of the project, StarPU is extended to support the Data Flow Engines (DFEs) architecture. Task scheduling is going to combine multiple scheduling criteria like data transfers and energy consumption. StarPU is also extended to support fault tolerance mechanisms, based mainly on data replication.

[5] Main webpage: `https://exa2pro.eu`

[6] At the time of writing, the project is still in progress. The end date is the 30 of April 2021

4. EVALUATION OF STARPU IN POLYBENCH BENCHMARK SUITE APPLICATIONS

Polybench[1] is a collection of benchmarks with static control flow extracted from operations in various application domains like linear algebra, image processing, etc. The benchmarks are availeble in CPU and GPU implementations (PolyBench/C 4.1 and PolyBench/GPU 1.0).

StarPU was applied to some Polybench benchmarks and two other applications in order to evaluate StarPU's scheduling decision-making in heterogeneous systems. The process followed was first to taskify the applications and implement them in StarPU. The next step was to apply one of StarPU's performance models[2] to the applications. The last step was to execute the applications in an heterogeneous node (CPU and GPU) for different input sizes and observe the scheduling decisions taken by StarPU.

The rest of the chapter consists of the description of each application and their results. During implementation, there was a test case that `starpu_task_ft_failed`[3] had to be used. This instruction has the restriction that all task data must have either read or write access mode and not read/write. Even though this instruction was, eventually, not used, the applications sum of matrix row elements, 2D convolution and 1D jacobi stencil computation were modified to have data only in read or write access mode.

4.1 Sum of matrix row elements

The sum of matrix row elements application is a simplified version of a count-based streaming aggregation algorithm, as described in [20]. The kernel sums the elements of the input array $window$. The equation that describes the computation is

$$sum = \sum_{i=0}^{n-1} window[i],$$

where `window` is a flattened 2D matrix of unsigned integers with 32 bits width and n number of elements and `sum` is the output, an unsigned integer variable with 64 bits width. The codelet has to manipulate two data buffers, one in read mode (window) and one in write mode (sum).

```
1 static struct starpu_codelet cl =
2 {
3   /*CPU implementation of the codelet */
4   .cpu_funcs = { cpu_output },
5   .cpu_funcs_name = { "cpu_output" },
6   #ifdef STARPU_USE_CUDA
```

[1] Main webpage: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`
[2] We decided to apply the History-based performance model.
[3] This instruction is currently included only on the git release

```
  /* CUDA implementation of the codelet */
  .cuda_funcs = { output_thread_aggregation },
  #endif
  .nbuffers = 2,
  .modes = { STARPU_R, STARPU_W },
  .model = &perf_model
};
```

Code 4.1: Codelet of sum of matrix row elements application

```
for (int i = 0; i < n; i++){
  cpu_sum += window[i];
}
```

Code 4.2: C implementation kernel of sum of matrix row elements

4.2 2D convolution

In the digital domain, 2D convolution is performed between two two-dimensional signals by multiplying and accumulating the values of the overlapping samples of the signals. In the 2D convolution application, the input matrix `A` is being convolved with the kernel matrix `c` and the result is stored in matrix `B`. The mathematical formulation that the kernel implements is

$$B[i][j] = \sum_{m=-1}^{1} \sum_{n=-1}^{1} c[m+1][n+1] \cdot A[i-m][j-n], i \in [1, n_i - 1], j \in [1, n_j - 1],$$

where `A` is an $n_i \times n_j$ matrix of floats, `c` is a 3×3 matrix of floats that is initialized inside the kernel as independent constants and `B` is the output matrix of floats. The codelet has to manipulate two data buffers (for the input and output matrices) in read and write mode respectively.

```
static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { conv2D },
  .cpu_funcs_name = { "conv2D" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { convolution2DCuda },
  #endif
  .nbuffers = 2,
  .modes = { STARPU_R, STARPU_W },
  .model = &perf_model
};
```

Code 4.3: Codelet of 2D convolution application

```
for (i = 1; i < ni - 1; ++i) { // 0
  for (j = 1; j < nj - 1; ++j) {  // 1
    B[(i*nj)+j] = c11 * A[((i - 1)*nj)+(j - 1)]  +  c12 * A[((i + 0)*nj)+(j - 1)]  +  c13 * A[((i + 1)*nj)+(j - 1)]
      + c21 * A[((i - 1)*nj)+(j + 0)]  +  c22 * A[((i + 0)*nj)+(j + 0)]  + c23 * A[((i + 1)*nj)+(j + 0)]
      + c31 * A[((i - 1)*nj)+(j + 1)]  +  c32 * A[((i + 0)*nj)+(j + 1)]  +  c33 * A[((i + 1)*nj)+(j + 1)];
  }
}
```

Code 4.4: C implementation kernel of 2D convolution

4.3 1D jacobi stencil computation

Stencil computations are a class of algorithms that repeatedly update values in a multi-dimensional grid using a fixed pattern of neighboring values, the stencil. For every input value a neighborhood that is specified by the stencil's shape is accessed. The elements of the neighborhood are used to compute an output value. Particular attention must be paid to border handling.

In the 1D jacobi stencil computation application, the equations that the kernel implements are

$$B[i] = 0.33333(A(i-1) + A(i) + A(i+1)), i \in [2, n-2]$$
$$A[i] = B[i], i \in [2, n-2]$$

The computation grid in the kernel is the $[0, n-1]$ region, where the stencil should be applied repeatedly over `tsteps` times. The application defines two input arrays (`A` and `B`) and two output arrays (`A_out` and `B_out`) of floats and n number of elements. In every iteration, array `B` caches the new values of `A` that are defined in the range $[2, n-2]$, and `A` contains the values of the previous iteration. When all the iterations are done, the final values are stored in the output arrays.

```
static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { jacobi1D_cpu },
  .cpu_funcs_name = { "jacobi1D_cpu" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { jacobi1D_gpu },
  #endif
  .nbuffers = 4,
  .modes = { STARPU_R, STARPU_R, STARPU_W, STARPU_W },
  .model = &perf_model
};
```

Code 4.5: Codelet of 1D jacobi stencil computation application

```
for (int t = 0; t < tsteps; t++) {
  for (int i = 2; i < n - 1; i++) {
    B_cpu[i] = 0.33333 * (A_cpu[i-1] + A_cpu[i] + A_cpu[i + 1]);
  }

  for (int j = 2; j < n - 1; j++) {
    A_cpu[j] = B_cpu[j];
  }
}

for (int i = 0; i < n; i++){
  A_out[i] = A_cpu[i];
  B_out[i] = B_cpu[i];
}
```

Code 4.6: C implementation kernel of 1D jacobi stencil computation

4.4 Matrix vector product and transpose

Matrix vector product is one of the most common kernels in linear algebra. In order to multiply a matrix A with a vector x, the number of columns of A must be equal to the number of rows of x (x is seen as a column matrix). If A is an $m \times n$ matrix, then the dimensions of x must be $n \times 1$. The result of the matrix vector product is an $m \times 1$ vector.

In this application, the elements of the matrix vector product between input matrix A and input vector y_1 are added to the elements of input/output vector x_1, and the elements of the matrix vector product between transpose matrix A and input vector y_2 are added to the elements of input/output vector x_2. The computations are described by the equations

$$x_1 \leftarrow x_1 + Ay_1$$
$$x_2 \leftarrow x_2 + A'y_2,$$

where x_1, x_2, y_1 and y_2 are arrays of floats of n elements and A is a matrix of floats of $n \times n$ elements. The codelet manipulates three data buffers of read mode and two data buffers of read/write mode.

```
static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { mvt_cpu },
  .cpu_funcs_name = { "mvt_cpu" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { mvt_cuda },
  #endif
  .nbuffers = 5,
  .modes = { STARPU_R, STARPU_RW, STARPU_RW, STARPU_R, STARPU_R },
  .model = &perf_model
};
```

Code 4.7: Codelet of matrix vector product and transpose application

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    x1[i] = x1[i] + A[(i*n)+j] * y1[j];
    }
}

for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    x2[i] = x2[i] + A[(j*n)+i] * y2[j];
  }
}
```

Code 4.8: C implementation kernel of matrix vector product and transpose

4.5 Symmetric rank-2K operations

Basic Linear Algebra Subprograms (BLAS)[4] are high quality routines that perform basic vector and matrix operations. BLAS are categorized in three levels, Level 1 for vector-vector operations, Level 2 for matrix-vector operations and Level 3 for matrix matrix operations.

Symmetric rank-2K operations are part of Level 3 BLAS. The mathematical formulation that the kernel implements is

$$C \leftarrow alphaAB' + alphaBA' + betaC,$$

where $alpha$, $beta$ are float scalars, A, B are input matrices of floats with dimensions $n_i \times n_j$ and C is an input/output $n_i \times n_i$ symmetric matrix. The scalars are passed to the kernel function through buffers, therefore the codelet has to manipulate five data buffers, four in read mode and one in read/write mode.

```
static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { syr2k_cpu },
  .cpu_funcs_name = { "syr2k_cpu" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { syr2k_cuda },
  #endif
  .nbuffers = 5,
  .modes = { STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_RW },
  .model = &perf_model
};
```

Code 4.9: Codelet of symmetric rank-2K operations application

[4] http://www.netlib.org/blas/

```
for (i = 0; i < ni; i++) {
  for (j = 0; j < ni; j++) {
    C[(i*ni)+j] *= *beta;
  }
}

for (i = 0; i < ni; i++) {
  for (j = 0; j < ni; j++) {
    for (k = 0; k < nj; k++) {
      C[(i*ni)+j] += *alpha * A[(i*nj)+k] * B[(j*nj)+k];
      C[(i*ni)+j] += *alpha * B[(i*nj)+k] * A[(j*nj)+k];
    }
  }
}
```

Code 4.10: C implementation kernel of symmetric rank-2K operations

4.6 Brain modeling

Brain modeling [19] is a C++ mini-application for neuron simulation and is very performance demanding. In [19] the miniapp was ported to SkePU to target multiple parallelization platforms. SkePU[5] [11] is a skeleton programming framework targeting multiple implementation variants (backends) and extends modern C++ with data-parallel skeletons.

In this application, StarPU was not directly applied in the code, but it was applied with the help of another EXA2PRO framework, ComPU. ComPU[6] [9] is a PEPPHER composition tool that collects information about the applications and its components from XML-based metadata descriptors, and generates glue code for the RTS. In the context of the EXA2PRO project, ComPU was modified as to use the EXA2PRO RTS, that is build upon StarPU's RTS. It's functionality was also extended [16].

The brain modeling application was ported to ComPU, that uses a multi-variant component with two implementations (CPU, GPU) for the computational core function. For the component operands, skepu data-containers (e.g skepu::Vector<>) were used.

The kernel functions of the application has six arguments. The codelet manipulates four data buffers, as two of the arguments are parameters and are handled by $task->cl\_arg$.

```
if(! objSt_directSim->cl_directSim_init ) // codelete initialization only once
    , at first invocation
 {
 objSt_directSim->cl_directSim.where =0|STARPU_CPU|STARPU_CUDA;

 objSt_directSim->cl_directSim.cpu_funcs[0]=directSim_cpu_wrapper;
objSt_directSim->cl_directSim.cpu_funcs[1]=NULL;

```

[5] Main webpage: `https://skepu.github.io`

[6] `https://www.ida.liu.se/labs/pelab/ctool/`

```
 objSt_directSim->cl_directSim.cuda_funcs[0]=directSim_cuda_wrapper;
objSt_directSim->cl_directSim.cuda_funcs[1]=NULL;

 objSt_directSim->cl_directSim.nbuffers = 4;

  objSt_directSim->cl_directSim.modes[0] = STARPU_W;
objSt_directSim->cl_directSim.modes[1] = STARPU_R;
objSt_directSim->cl_directSim.modes[2] = STARPU_R;
objSt_directSim->cl_directSim.modes[3] = STARPU_R;

 objSt_directSim->cl_directSim_init = 1;
 }
```

Code 4.11: Codelet of brain modeling application (part of the wrapper file generated by ComPU)

```
for (int i = 0; i < rows; ++i){

    // get context for i-th neuron
    Constants parms = constants(i);
    State statePrev = state(i);

    // get currents from gap junctions
    float iGapTotal = 0;
    for (int j = 0; j < cols; ++j){
      State sti = state(i);
      State stj = state(j);
      iGapTotal += fGap(m(i, j) , sti.v, stj.v); // fGap( m[i*cols+j] , sti.v,
    stj.v );
    }

    // get next state
    State stateNext = InnerDynamics_Integrate(parms, statePrev, iGapTotal,
   time, dt);

    res(i) = stateNext;
  }
```

Code 4.12: C implementation kernel of brain modeling

4.7 Result evaluation

The measurements for the applications were conducted in different computing systems. For the applications sum of matrix row elements, 2D convolution and 1D jacobi stencil operations the measurements were taken in a computing system with a 2x Intel Xeon Gold 6138 CPU (2x20 H/T cores) and an NVIDIA Tesla V-100 GPU. For the rest of the applications, the measurements were conducted in a computing system with a Xeon E5-2630L CPU (12 cores) and an NVIDIA K20c GPU.

All applications, except of brain modeling, consist of one task that is being submitted just one time. In contrast, brain modeling consist of one task that is being submitted 20

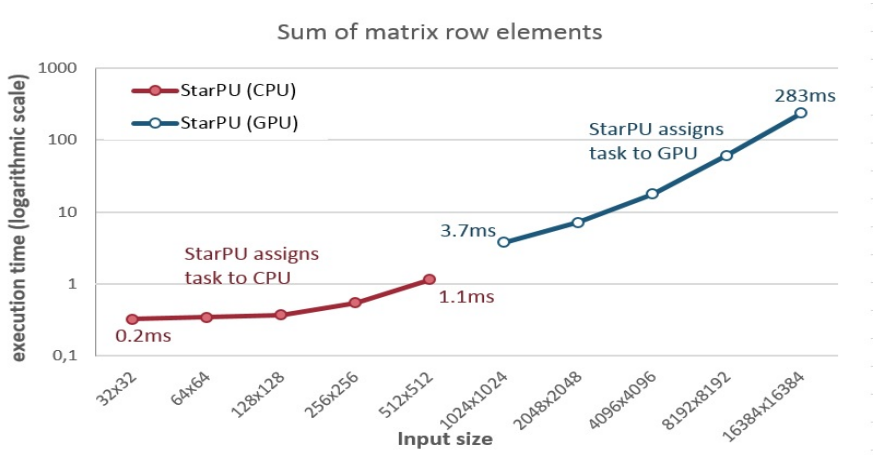


Figure 4.1: Sum of matrix row elements scheduling decisions

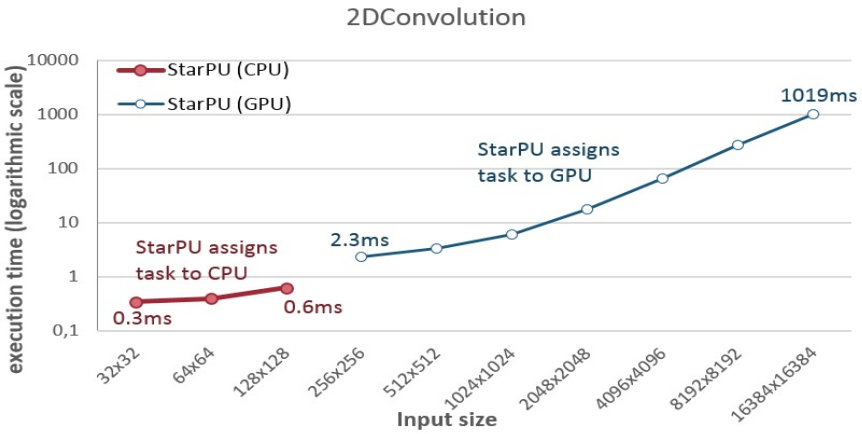


Figure 4.2: 2D convolution scheduling decisions

times[7]. Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 show StarPU's scheduling decisions for the applications. The scheduling decisions were based on the execution time of the kernels on the different processing units, as we used the history-based performance model. In the graphs, x axis corresponds to the input sizes of the applications (or the number of neurons in the case of the brain modeling application) and y axis to the execution time in logarithmic scale. From the figures mentioned above, we observe that StarPU assigned small computational workload tasks to CPU, whereas large computational workload tasks to GPU.

Figures 4.7a, 4.7b and 4.7c represent the execution time of the non StarPU-ized CPU implementation and the execution time of the StarPU-ized version of the sum of matrix row elements, 2D convolution and jacobi 1D applications respectively. For the StarPU-ized version, the execution times are the ones measured after StarPU's scheduling decisions. From these figures, we notice that for small input sizes, the execution time of the StarPU-ized applications is worse than the non StarPU-ized, due to the runtime system's

[7]Brain modeling was executed with $steps = 20$ and $density = 0.12$.

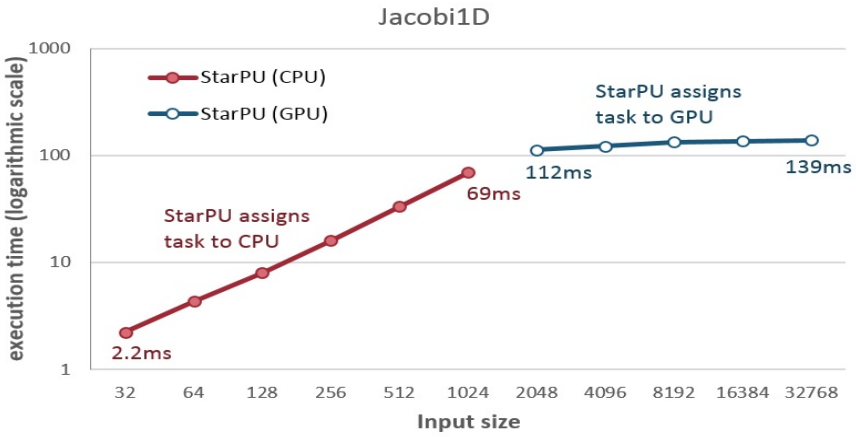


Figure 4.3: 1D jacobi stencil operation scheduling decisions

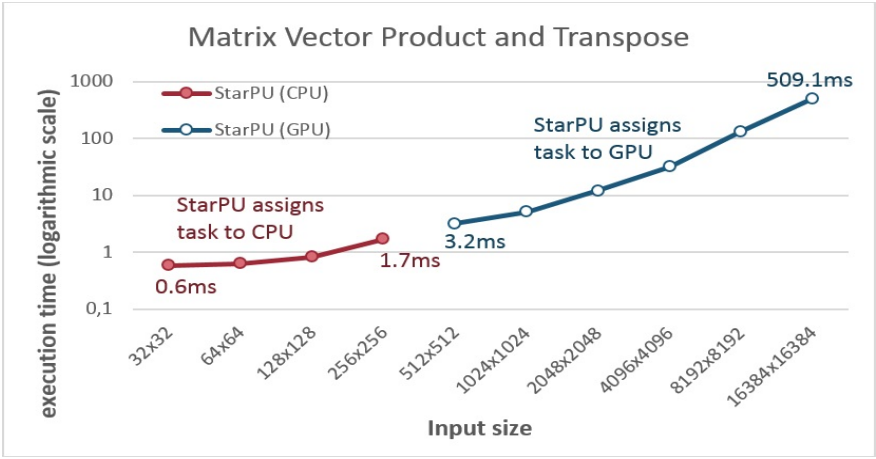


Figure 4.4: Matrix vector product and transpose scheduling decisions

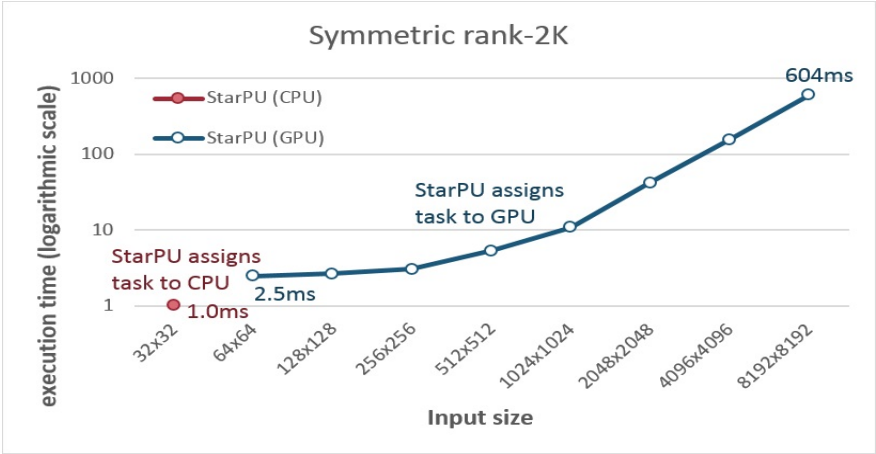


Figure 4.5: Symmetric rank-2D operations scheduling decisions

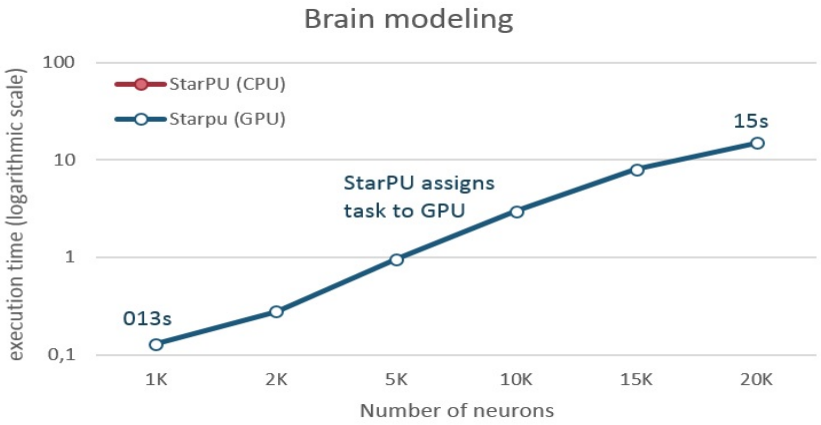


Figure 4.6: Brain modeling scheduling decisions

overhead.

Figure 4.7d depicts the execution times of the StarPU-ized version of the brain modeling application for the CPU and GPU implementations, as well as for the StarPU's selection of a processing unit to execute the codelet. From this figure, we can discern that the execution time of the CPU implementation is greater than the GPU's implementation. In addition, the execution time of StarPU's selection is slightly less than the one of the GPU implementation. This is due to the data prefetch the `dmda`[8] scheduler performs. That means that the task's required data are requested to get transferred to the target processing unit once a scheduling decision in taken. As a result, the task does not have to wait for the data transfer to the target processing unit to finish, as data are already available.

4.8 Training/programming effort

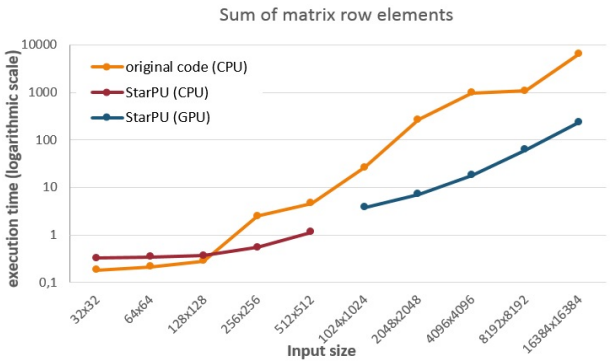
I had no prior experience in StarPU before my engagement in this master thesis. In order to get familiar with StarPU, there was a training period of about two months that offered limited familiarization.

As already mentioned, StarPU was applied to some applications that I was coming in contact with their source code for the first time. This fact, in combination with the limited familiarization to StarPU, lead to a period of time of about two weeks to apply the steps for evaluating StarPU's scheduling decisions to the first application, sum of matrix row elements. For the Polybench applications, the time to evaluate StarPU was reduced to two days[9].

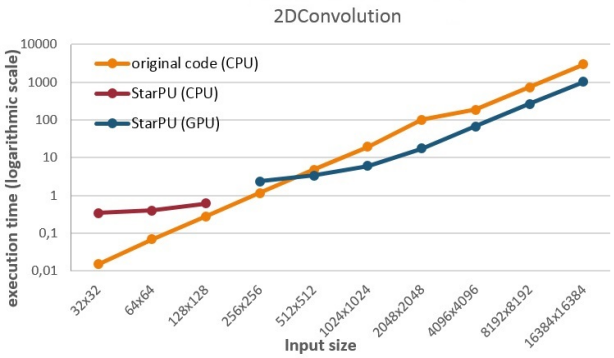
In every application, the number of StarPU-specific lines of source code depends on dif-

[8]The `dmda` scheduler (see Section 3.3) is the one we chose for being able to make use of the history-based performance model.

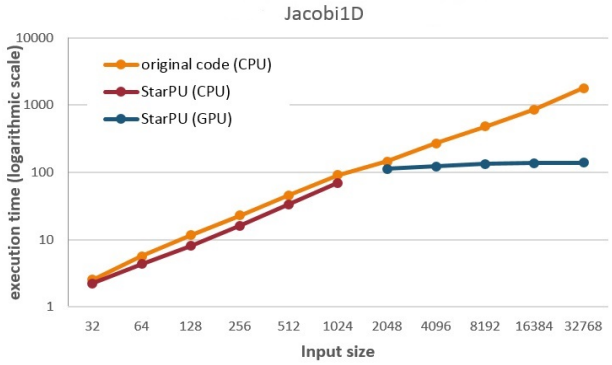
[9]The brain modeling is not included in this Section, because StarPU was applied in a different way that required training for a different framework.



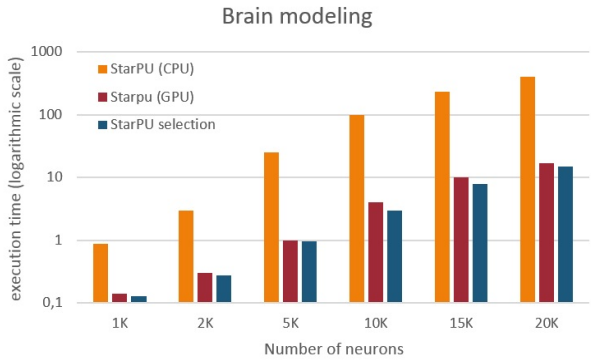
(a) Sum of matrix row elements execution time of original code (CPU implementation) and StarPU selection



(b) 2D convolution execution time of original code (CPU implementation) and StarPU selection



(c) 1D jacobi stencil operation execution time of original code (CPU implementation) and StarPU selection



(d) Brain modeling execution time of StarPU-ized CPU implementation, StarPU-ized GPU implementation and StarPU selection

Figure 4.7: Comparison of execution time of original code and StarPU-ized

ferent factors, like the available implementations of the kernel, the number of data buffers that the codelet has to manipulate, etc. The total number of source code lines needed in each application for initializing and terminating StarPU, defining data handlers, submitting tasks, etc was about 35.

4.9 Discussion

From the experiments conducted, we conclude that with StarPU is easier to implement an application for different processing units, as the only thing that has to be implemented differently is the kernel. In addition, StarPU-ized applications perform well in comparison to non StarPU-ized applications. However, for small input sizes is preferable not to apply StarPU due to the runtime system's overhead.

The execution time of a kernel on a specific processing unit is an important parameter that affects StarPU's scheduling decisions. Input size and, as a consequence, data transfer time are important factors that affect the execution time of the kernel. Another factor is the complexity of the kernel and how efficiently is written. From the obtained results we

observe that StarPU takes the appropriate scheduling decision, by taking into account where the task would be executed more efficiently.

From our experience, we estimate that the required time to get to use StarPU, for a user unfamiliar with the specific framework, is about two months. On the other hand, we cannot estimate the time required to apply StarPU in an application as it depends on different factors. These factors are the complexity of the application and how familiar is the programmer with the source code of the application.

An important factor for evaluating task-based runtime systems is scalability. In [27] StarPU's scalability was evaluated. The work was conducted at the Institute of Communications and Computer Systems (ICCS) and at the Centre National de la Recherche Scientifique (CNRS). The results were controversial as the application used by ICCS demonstrated not so good scalability, but the application used by CNRS demonstrated good scalability. According to the research, scalability is affected by the task size, which is hard to be specified by the programmer. This creates the need for more research on how task-based runtime systems could be more efficient for exascale systems.

Even though the above comments are for StarPU, we assume that similar conclusion can be drawn when using other task-based RTSs.

5. CONCLUSIONS AND FUTURE WORK

In this master thesis some task-based parallel models were presented, with more emphasis being given to StarPU. The reason for deciding to study task-based parallel models is that they consist a promising solution to the challenges related to nowadays manycore heterogeneous architectures.

The object of the thesis was to observe the scheduling decisions of StarPU in heterogeneous systems. For that purpose, StarPU was applied to some applications (mostly Polybench benchmarks). More specifically, we taskified and implemented the available applications codes in StarPU and executed the applications in an heterogeneous node for different input sizes. StarPU's decision-making was based on the history-based performance model that was applied to the applications.

From the obtained results, we observed that StarPU took the most appropriate scheduling decisions and assigned tasks with small workloads to CPU and with large workloads to GPU. The decisions were taken based on the execution time of the kernels on each processing unit, where the execution time was affected by different factors, such as data transfer time.

This thesis could be extended in various ways. Firstly, it would be interesting to add more implementations of the applications, like OpenCL, and observe StarPU's scheduling decisions with more than two implementations. In addition, we could test if the scheduling decisions would be different with the application of another performance model. Moreover, we could apply StarPU in more complex applications. Lastly, we could extend the applications as to include more StarPU features, like fault tolerance support.

ABBREVIATIONS - ACRONYMS

BLAS	Basic Linear Algebra Subprograms
CNRS	Centre National de la Recherche Scientifique
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DFE	Data Flow Engine
EXA2PRO	Enhancing Programmability and boosting Performance Portability for Exascale Computing Systems
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HPX	High Performance ParallelX
ICCS	Institute of Communications and Computer Systems
ILP	Instruction Level Parallelism
MPI	Message Passing Interface
PRAM	Parallel Random Access Machine
RTS	Runtime System
STF	Sequential Task Flow
TBB	Threading Building Blocks
TDG	Task Dependence Graph

APPENDIX A. APPLICATIONS CODES

Code A.1: Sum of matrix row elements main code

```
#include <starpu.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <stdint.h>

// The window height is equal to the number of streams
#define WINDOW_HEIGHT   32
// The window width is equal to number of tuples required to fill in a window
#define WINDOW_WIDTH    32

// The total number of elements is equal to the window_height times
    window_width
#define elements      WINDOW_HEIGHT*WINDOW_WIDTH

#define FPRINTF(ofile, fmt, ...) do { if (!getenv("STARPU_SSILENT")) {fprintf(
    ofile, fmt, ## __VA_ARGS__); }} while(0)

// measure time:
struct timeval start, end;

extern void cpu_output(void *descr[], void *_args);
extern void output_thread_aggregation(void *descr[], void *_args);

static struct starpu_perfmodel perf_model = {
  .type = STARPU_HISTORY_BASED,
  .symbol = "main_32",
};

static struct starpu_codelet cl =
{
  /*CPU implementation of the codelet */
  .cpu_funcs = { cpu_output },
  .cpu_funcs_name = { "cpu_output" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { output_thread_aggregation },
  #endif
  .nbuffers = 2,
  .modes = { STARPU_R, STARPU_W },
  .model = &perf_model
};

int main(int argc, char **argv)
{

  // dynamic allocation of the memory needed for all the elements
  uint32_t *window;
  window = (uint32_t*)calloc(elements, sizeof(uint32_t));
```

```
48
49  // check if there's enough space for the allocation
50  if(!window){
51    printf("Allocation error for window - aborting.\n");
52    exit(1);
53  }
54
55  // initialization - fill in the window with random numbers:
56  for (int i = 0; i < elements; i++) {
57    window[i] = (rand()%1000);
58  }
59
60  /* Initialize StarPU with default configuration */
61  int ret = starpu_init(NULL);
62  if (ret == -ENODEV) goto enodev;
63
64  /* initialize performance model */
65  starpu_perfmodel_init(&perf_model);
66
67  uint64_t aggregated_value = 1;
68
69  starpu_data_handle_t vector_handle, variable_handle;
70
71  /* Tell StaPU to associate the "window" vector with the "vector_handle" */
72  starpu_vector_data_register(&vector_handle, STARPU_MAIN_RAM, (uintptr_t)
     window, elements, sizeof(window[0]));
73
74  /* Tell StarPU to associate the "aggregated_value" (variable where the
     codelet will store its result) with the "variable_handle" */
75  starpu_variable_data_register(&variable_handle, STARPU_MAIN_RAM, (uintptr_t)
     &aggregated_value, sizeof(aggregated_value));
76
77  gettimeofday(&start, NULL);
78
79  starpu_task_insert( &cl,
80            STARPU_R, vector_handle,
81            STARPU_W, variable_handle,
82            0);
83
84  /* StarPU does not need to manipulate the array and the variables anymore so
       we can stop monitoring them */
85  starpu_data_unregister(vector_handle);
86  starpu_data_unregister(variable_handle);
87
88  gettimeofday(&end, NULL);
89
90  /* terminate StarPU */
91  starpu_shutdown();
92
93  fprintf(stderr,"aggregated value: %lu \n", aggregated_value);
94
95  //free the memory allocated on the CPU
96  free(window);
```

```
97
98    // calculate the time required for the calculation of the aggregated value
99    double elapsed = (end.tv_sec*1000000+end.tv_usec)-(start.tv_sec*1000000+
        start.tv_usec);
100   fprintf(stderr, "Elapsed time in application is: %lf msec\n", (elapsed)
        /1000.0F);
101
102 enodev:
103   return 77;
104 }
```

Code A.2: Sum of matrix row elements CPU kernel implementation

```
1  #include <starpu.h>
2
3  void cpu_output(void *descr[], void *_args)
4  {
5    printf("cpu function\n");
6
7    /* length of the vector */
8    int n = STARPU_VECTOR_GET_NX(descr[0]);
9
10   /* local copy of the vector and variable pointers */
11   uint32_t *window = (uint32_t *)STARPU_VECTOR_GET_PTR(descr[0]);
12   uint64_t *aggregated_value = (uint64_t *)STARPU_VARIABLE_GET_PTR(descr[1]);
13
14   uint64_t cpu_sum = 0;
15
16   /* compute the aggregation value */
17   for (int i = 0; i < n; i++) {
18     cpu_sum += window[i];
19   }
20
21   *aggregated_value = cpu_sum;
22 }
```

Code A.3: Sum of matrix row elements GPU kernel implementation

```
1  #include <stdio.h>
2  #include <starpu.h>
3  #include <math.h>
4
5  __global__ void output_thread_aggregation(uint32_t *window, uint32_t *dev_data
       , int n)
6  {
7    uint tid = threadIdx.x;
8    uint index = blockIdx.x*blockDim.x + threadIdx.x;
9
10   // convert global data pointer to the local pointer of the block
11   uint32_t *idata = window + blockIdx.x * blockDim.x;
12
13   if (index >= n) return;
14
15   // reduction algorithm
```

```
16    for (int stride = blockDim.x/2; stride > 0; stride>>=1){
17      if (tid < stride){
18        idata[tid] += idata[tid+stride];
19      }
20      __syncthreads();
21    }
22
23    // write result for this block to global memory
24    if(tid == 0){
25      dev_data[blockIdx.x] = idata[0];
26    }
27
28    for (int i=0; i<tid; i++){
29      idata[tid] = 0;
30    }
31  }
32
33  __global__ void result(uint64_t *aggregated_value, uint64_t *dev_sum)
34  {
35    *aggregated_value = *dev_sum;
36  }
37
38  extern "C" void output_thread_aggregation(void *descr[], void *_args)
39  {
40    printf("cuda function\n");
41
42    /* length of the vector */
43    int n = STARPU_VECTOR_GET_NX(descr[0]);
44
45    /* local copy of the vector and variable pointers */
46    uint32_t *window = (uint32_t *)STARPU_VECTOR_GET_PTR(descr[0]);
47    uint64_t *aggregated_value = (uint64_t *)STARPU_VARIABLE_GET_PTR(descr[1]);
48
49    /* define the number of threads per block accordingly to the vector's size
       */
50    int n_threads;
51    if (sqrt(n) <= 32)
52      {n_threads = 32;}
53    else if (sqrt(n) <= 64)
54      {n_threads = 64;}
55    else if (sqrt(n) <= 128)
56      {n_threads = 128;}
57    else if (sqrt(n) <= 256)
58      {n_threads = 256;}
59    else if (sqrt(n) <= 512)
60      {n_threads = 512;}
61    else
62      {n_threads = 1024;}
63
64    // define number of blocks and number of threads per block (kernel
       parameters)
65    dim3 threads_per_block (n_threads);
66    dim3 blocks ((n+threads_per_block.x-1)/threads_per_block.x);
```

```
67
68    // dynamic allocation of the reduced data matrix
69    uint32_t *h_data = (uint32_t *)malloc(blocks.x*sizeof(uint32_t));
70
71    if(!h_data){
72      printf("Allocation error for h_data - aborting.\n");
73      exit(1);
74    }
75
76    // GPU memory pointers
77    uint32_t *dev_data;
78    uint64_t *dev_sum;
79
80    // allocate the memory on the GPU
81    cudaMalloc((void**)&dev_data, blocks.x*sizeof(uint32_t));
82    cudaMalloc((void**)&dev_sum, sizeof(uint64_t));
83
84    // launch kernel
85    output_thread_aggregation<<<blocks,threads_per_block>>>(window, dev_data, n)
        ;
86
87    cudaStreamSynchronize(starpu_cuda_get_local_stream());
88
89    // copy back the result to the CPU
90    cudaMemcpyAsync(h_data, dev_data, blocks.x*sizeof(uint32_t),
        cudaMemcpyDeviceToHost);
91
92    uint64_t gpu_sum = 0;
93    uint64_t *p = &gpu_sum;
94
95    // compute the total sum from gpu
96    for(int i=0; i<blocks.x; i++){
97      gpu_sum += h_data[i];
98    }
99
100   cudaMemcpy(dev_sum, p, sizeof(uint64_t), cudaMemcpyHostToDevice);
101   result<<<1,1>>>(aggregated_value, dev_sum);
102
103   //free the memory allocated on the GPU
104   cudaFree(dev_data);
105   cudaFree(dev_sum);
106 }
```

Code A.4: 2D convolution main code

```
1  /**
2   * 2DConvolution.cu: This file is part of the PolyBench/GPU 1.0 test suite.
3   *
4   *
5   * Contact: Scott Grauer-Gray <sgrauerg@gmail.com>
6   * Will Killian <killian@udel.edu>
7   * Louis-Noel Pouchet <pouchet@cse.ohio-state.edu>
8   * Web address: http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
9   */
```

```
10
11 #include <unistd.h>
12 #include <stdio.h>
13 #include <time.h>
14 #include <sys/time.h>
15 #include <stdlib.h>
16 #include <stdarg.h>
17 #include <string.h>
18
19 #include <starpu.h>
20
21 #define NI 32
22 #define NJ 32
23
24 #define FPRINTF(ofile, fmt, ...) do { if (!getenv("STARPU_SSILENT")) {fprintf(
       ofile, fmt, ## __VA_ARGS__); }} while(0)
25
26 // measure time:
27 struct timeval start, end;
28
29 extern void conv2D(void *descr[], void *_args);
30 extern void convolution2DCuda(void *descr[], void *_args);
31
32 static struct starpu_perfmodel perf_model = {
33   .type = STARPU_HISTORY_BASED,
34   .symbol = "2DConv_32",
35 };
36
37 static struct starpu_codelet cl =
38 {
39   /*CPU implementation of the codelet */
40   .cpu_funcs = { conv2D },
41   .cpu_funcs_name = { "conv2D" },
42   #ifdef STARPU_USE_CUDA
43   /* CUDA implementation of the codelet */
44   .cuda_funcs = { convolution2DCuda },
45   #endif
46   .nbuffers = 2,
47   .modes = { STARPU_R, STARPU_W },
48   .model = &perf_model
49 };
50
51 void init(int ni, int nj, float *A)
52 {
53   int i, j;
54
55   for (i = 0; i < ni; ++i){
56     for (j = 0; j < nj; ++j){
57       A[(i*nj)+j] = (float)rand()/RAND_MAX;
58         }
59     }
60 }
61
```

```
62 void print_array(int ni, int nj, float *B)
63 {
64   int i, j;
65
66   for (i = 0; i < ni; i++){
67     for (j = 0; j < nj; j++) {
68       fprintf (stderr, "%0.2lf ", B[(i*nj)+j]);
69       if ((i * ni + j) % 20 == 0) fprintf (stderr, "\n");
70     }
71   }
72   fprintf (stderr, "\n");
73 }
74
75 int main(int argc, char **argv)
76 {
77   /* Retrieve problem size */
78   int ni = NI;
79   int nj = NJ;
80
81   float *A = (float *)malloc(NI*NJ*sizeof(float));
82   float *B = (float *)malloc(NI*NJ*sizeof(float));
83
84   // check if there's enough space for the allocation
85   if(!A){
86     printf("Allocation error for A - aborting.\n");
87     exit(1);
88   }
89
90   if(!B){
91     printf("Allocation error for B - aborting.\n");
92     exit(1);
93   }
94
95   //initialize the arrays
96   init(ni, nj, A);
97
98   /* Initialize StarPU with default configuration */
99   int ret = starpu_init(NULL);
100  if (ret == -ENODEV) goto enodev;
101
102  /* initialize performance model */
103  starpu_perfmodel_init(&perf_model);
104
105  starpu_data_handle_t matrixa_handle, matrixb_handle;
106  starpu_matrix_data_register(&matrixa_handle, STARPU_MAIN_RAM, (uintptr_t)A,
       NI, NI, NJ, sizeof(A[0]));
107  starpu_matrix_data_register(&matrixb_handle, STARPU_MAIN_RAM, (uintptr_t)B,
       NI, NI, NJ, sizeof(B[0]));
108
109  gettimeofday(&start, NULL);
110
111  starpu_task_insert( &cl,
112            STARPU_R, matrixa_handle,
```

```
            STARPU_W, matrixb_handle,
            0);

  /* StarPU does not need to manipulate the array and the variables anymore so
    we can stop monitoring them */
  starpu_data_unregister(matrixa_handle);
  starpu_data_unregister(matrixb_handle);

  gettimeofday(&end, NULL);

  /* terminate StarPU */
  starpu_shutdown();

  // print_array(ni, nj, B);

  //free the memory allocated on the CPU
  free(A);
  free(B);

  // calculate the time required for the calculation
  double elapsed = (end.tv_sec*1000000+end.tv_usec)-(start.tv_sec*1000000+
   start.tv_usec);
  fprintf(stderr, "Elapsed time in application is: %lf msec\n", (elapsed)
   /1000.0F);

  enodev:
    return 77;
}
```

Code A.5: 2D convolution CPU kernel implementation

```
#include <starpu.h>

void conv2D(void *descr[], void *_args)
{
  printf("cpu function\n");

  /* length of the matrix */
  unsigned ni = STARPU_MATRIX_GET_NX(descr[0]);
  unsigned nj = STARPU_MATRIX_GET_NY(descr[0]);

  /* local copy of the matrix pointer */
  float *A = (float *)STARPU_MATRIX_GET_PTR(descr[0]);
  float *B = (float *)STARPU_MATRIX_GET_PTR(descr[1]);

  int i, j;
  float c11, c12, c13, c21, c22, c23, c31, c32, c33;


  c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
  c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
  c13 = +0.4;  c23 = +0.7;  c33 = +0.10;

  for (i = 1; i < ni - 1; ++i){ // 0
```

```
    for (j = 1; j < nj - 1; ++j){ // 1
      B[(i*nj)+j] = c11 * A[((i - 1)*nj)+(j - 1)]  +  c12 * A[((i + 0)*nj)+(j - 1)]  +  c13 * A[((i + 1)*nj)+(j - 1)]
        + c21 * A[((i - 1)*nj)+(j + 0)]  +  c22 * A[((i + 0)*nj)+(j + 0)]  +  c23 * A[((i + 1)*nj)+(j + 0)]
        + c31 * A[((i - 1)*nj)+(j + 1)]  +  c32 * A[((i + 0)*nj)+(j + 1)]  +  c33 * A[((i + 1)*nj)+(j + 1)];
    }
  }
}
```

Code A.6: 2D convolution GPU kernel implementation

```
#include <starpu.h>

/* Thread block dimensions */
#define DIM_THREAD_BLOCK_X 32
#define DIM_THREAD_BLOCK_Y 8

__global__ void convolution2D_kernel(int ni, int nj, float *A, float *B)
{
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;

  float c11, c12, c13, c21, c22, c23, c31, c32, c33;

  c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
  c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
  c13 = +0.4;  c23 = +0.7;  c33 = +0.10;

  if ((i < ni-1) && (j < nj-1) && (i > 0) && (j > 0)){
    B[i * nj + j] =  c11 * A[(i - 1) * nj + (j - 1)]  + c21 * A[(i - 1) * nj + (j + 0)] + c31 * A[(i - 1) * nj + (j + 1)]
      + c12 * A[(i + 0) * nj + (j - 1)]  + c22 * A[(i + 0) * nj + (j + 0)] + c32 * A[(i + 0) * nj + (j + 1)]
      + c13 * A[(i + 1) * nj + (j - 1)]  + c23 * A[(i + 1) * nj + (j + 0)] + c33 * A[(i + 1) * nj + (j + 1)];
  }

  __syncthreads();

}


extern "C" void convolution2DCuda(void *descr[], void *_args)
{
  printf("cuda function\n");

  /* length of the matrix */
  unsigned ni = STARPU_MATRIX_GET_NX(descr[0]);
  unsigned nj = STARPU_MATRIX_GET_NY(descr[0]);

  /* local copy of the matrix pointer */
  float *A = (float *)STARPU_MATRIX_GET_PTR(descr[0]);
```

```
  float *B = (float *)STARPU_MATRIX_GET_PTR(descr[1]);

  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
  dim3 grid((size_t)ceil( ((float)ni) / ((float)block.x) ), (size_t)ceil( ((float)nj) / ((float)block.y)) );

  convolution2D_kernel <<< grid,block >>> (ni, nj, A, B);

  cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Code A.7: 1D jacobi stencil computation main code

```
/**
 * jacobi1D.cu: This file is part of the PolyBench/GPU 1.0 test suite.
 *
 *
 * Contact: Scott Grauer-Gray <sgrauerg@gmail.com>
 * Will Killian <killian@udel.edu>
 * Louis-Noel Pouchet <pouchet@cse.ohio-state.edu>
 * Web address: http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
 */

#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <math.h>

#include <starpu.h>

#define N   32

#define FPRINTF(ofile, fmt, ...) do { if (!getenv("STARPU_SSILENT")) {fprintf(ofile, fmt, ## __VA_ARGS__); }} while(0)

// measure time:
struct timeval start, end;

extern void jacobi1D_cpu(void *descr[], void *_args);
extern void jacobi1D_gpu(void *descr[], void *_args);

int tsteps = 10000;

static struct starpu_perfmodel perf_model = {
  .type = STARPU_HISTORY_BASED,
  .symbol = "jacobi1D",
};

static struct starpu_codelet cl =
{
```

```
  /*CPU implementation of the codelet */
  .cpu_funcs = { jacobi1D_cpu },
  .cpu_funcs_name = { "jacobi1D_cpu" },
  #ifdef STARPU_USE_CUDA
  /* CUDA implementation of the codelet */
  .cuda_funcs = { jacobi1D_gpu },
  #endif
  .nbuffers = 4,
  .modes = { STARPU_R, STARPU_R, STARPU_W, STARPU_W },
  .model = &perf_model
};

void init_array(int n, float *A, float *B)
{
  int i;

  for (i = 0; i < n; i++) {
    A[i] = ((float) 4 * i + 10) / n;
    B[i] = ((float) 7 * i + 11) / n;
    }
}

/* DCE code. Must scan the entire live-out data.
   Can be used also to check the correctness of the output. */
static
void print_array(int n, float *A)
{
  int i;

  for (i = 0; i < n; i++) {
    fprintf(stderr, "%0.2lf ", A[i]);
    if (i % 20 == 0) fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
}

int main(int argc, char** argv)
{
  extern int tsteps;

  /* Retrieve problem size. */
  int n = N;

  float *A = (float *)malloc(n*sizeof(float));
  float *B = (float *)malloc(n*sizeof(float));
  float *A_out = (float *)malloc(n*sizeof(float));
  float *B_out = (float *)malloc(n*sizeof(float));

  // check if there's enough space for the allocation
  if(!A){
    printf("Allocation error for A - aborting.\n");
    exit(1);
  }
```

```
94
95  if(!B){
96    printf("Allocation error for B - aborting.\n");
97    exit(1);
98  }
99
100 if(!A_out){
101   printf("Allocation error for A_out - aborting.\n");
102   exit(1);
103 }
104
105 if(!B_out){
106   printf("Allocation error for B_out - aborting.\n");
107   exit(1);
108 }
109
110 init_array(n, A, B);
111
112 /* Initialize StarPU with default configuration */
113 int ret = starpu_init(NULL);
114 if (ret == -ENODEV) goto enodev;
115
116 /* initialize performance model */
117 starpu_perfmodel_init(&perf_model);
118
119 starpu_data_handle_t a_handle, b_handle, aout_handle, bout_handle;
120 starpu_vector_data_register(&a_handle, STARPU_MAIN_RAM, (uintptr_t)A, n,
     sizeof(A[0]));
121 starpu_vector_data_register(&b_handle, STARPU_MAIN_RAM, (uintptr_t)B, n,
     sizeof(B[0]));
122 starpu_vector_data_register(&aout_handle, STARPU_MAIN_RAM, (uintptr_t)A_out,
       n, sizeof(A_out[0]));
123 starpu_vector_data_register(&bout_handle, STARPU_MAIN_RAM, (uintptr_t)B_out,
       n, sizeof(B_out[0]));
124
125 gettimeofday(&start, NULL);
126
127 starpu_task_insert( &cl,
128           STARPU_R, a_handle,
129           STARPU_R, b_handle,
130           STARPU_W, aout_handle,
131           STARPU_W, bout_handle,
132           0);
133
134 /* StarPU does not need to manipulate the vectors anymore so we can stop
     monitoring them */
135 starpu_data_unregister(a_handle);
136 starpu_data_unregister(b_handle);
137 starpu_data_unregister(aout_handle);
138 starpu_data_unregister(bout_handle);
139
140 gettimeofday(&end, NULL);
141
```

```
142   /* terminate StarPU */
143   starpu_shutdown();
144
145   // print_array(n, A_out);
146   // print_array(n, B_out);
147
148   free(A);
149   free(A_out);
150   free(B);
151   free(B_out);
152
153   // calculate the time required for the calculation
154   double elapsed = (end.tv_sec*1000000+end.tv_usec)-(start.tv_sec*1000000+
        start.tv_usec);
155   fprintf(stderr, "Elapsed time in application is: %lf msec\n", (elapsed)
        /1000.0F);
156
157   enodev:
158     return 77;
159 }
```

Code A.8: 1D jacobi stencil computation CPU kernel implementation

```
1  #include <starpu.h>
2
3  extern int tsteps;
4
5  void jacobi1D_cpu(void *descr[], void *_args)
6  {
7    printf("cpu function\n");
8
9    /* length of the vector */
10   unsigned n = STARPU_VECTOR_GET_NX(descr[0]);
11
12   /* local copy of the vectors and variable pointers */
13   float *A = (float *)STARPU_VECTOR_GET_PTR(descr[0]);
14   float *B = (float *)STARPU_VECTOR_GET_PTR(descr[1]);
15   float *A_out = (float *)STARPU_VECTOR_GET_PTR(descr[2]);
16   float *B_out = (float *)STARPU_VECTOR_GET_PTR(descr[3]);
17
18   float A_cpu[n], B_cpu[n];
19
20   for (int i = 0; i < n; i++){
21     A_cpu[i] = A[i];
22     B_cpu[i] = B[i];
23   }
24
25   for (int t = 0; t < tsteps; t++){
26     for (int i = 2; i < n - 1; i++){
27       B_cpu[i] = 0.33333 * (A_cpu[i-1] + A_cpu[i] + A_cpu[i + 1]);
28     }
29
30     for (int j = 2; j < n - 1; j++){
31       A_cpu[j] = B_cpu[j];
```

```
    }
    }

  for (int i = 0; i < n; i++){
    A_out[i] = A_cpu[i];
    B_out[i] = B_cpu[i];
  }
}
```

Code A.9: 1D jacobi stencil computation GPU kernel implementation

```
#include <starpu.h>

/* Thread block dimensions */
#define DIM_THREAD_BLOCK_X 256
#define DIM_THREAD_BLOCK_Y 1

extern int tsteps;

__global__ void runJacobiCUDA_kernel1(int n, float *Agpu, float *Bgpu)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if ((i > 1) && (i < (n-1))){
    Bgpu[i] = 0.33333f * (Agpu[i-1] + Agpu[i] + Agpu[i + 1]);
  }

  __syncthreads();
}

__global__ void runJacobiCUDA_kernel2(int n, float *Agpu, float *Bgpu)
{
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  if ((j > 1) && (j < (n-1))){
    Agpu[j] = Bgpu[j];
  }
}

extern "C" void jacobi1D_gpu(void *descr[], void *_args)
{
  printf("cuda function\n");

  /* length of the vector */
  unsigned n = STARPU_VECTOR_GET_NX(descr[0]);

  /* local copy of the vectors and variable pointers */
  float *A = (float *)STARPU_VECTOR_GET_PTR(descr[0]);
  float *B = (float *)STARPU_VECTOR_GET_PTR(descr[1]);
  float *A_out = (float *)STARPU_VECTOR_GET_PTR(descr[2]);
  float *B_out = (float *)STARPU_VECTOR_GET_PTR(descr[3]);

  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
  dim3 grid((unsigned int)ceil( ((float)n) / ((float)block.x) ), 1);
```

```
44
45    float* Agpu;
46    float* Bgpu;
47
48    cudaMalloc(&Agpu, n * sizeof(float));
49    cudaMalloc(&Bgpu, n * sizeof(float));
50
51    cudaMemcpy(Agpu, A, n * sizeof(float), cudaMemcpyHostToDevice);
52    cudaMemcpy(Bgpu, B, n * sizeof(float), cudaMemcpyHostToDevice);
53
54    for (int t = 0; t < tsteps ; t++){
55      runJacobiCUDA_kernel1 <<< grid, block >>> (n, Agpu, Bgpu);
56      cudaStreamSynchronize(starpu_cuda_get_local_stream());
57      runJacobiCUDA_kernel2 <<< grid, block>>> (n, Agpu, Bgpu);
58      cudaStreamSynchronize(starpu_cuda_get_local_stream());
59    }
60
61    cudaMemcpy(A_out, Agpu, sizeof(float) * n, cudaMemcpyDeviceToHost);
62    cudaMemcpy(B_out, Bgpu, sizeof(float) * n, cudaMemcpyDeviceToHost);
63
64    cudaFree(Agpu);
65    cudaFree(Bgpu);
66  }
```

Code A.10: Matrix vector prodict and transpose main code

```
1  /**
2   * mvt.cu: This file is part of the PolyBench/GPU 1.0 test suite.
3   *
4   *
5   * Contact: Scott Grauer-Gray <sgrauerg@gmail.com>
6   * Will Killian <killian@udel.edu>
7   * Louis-Noel Pouchet <pouchet@cse.ohio-state.edu>
8   * Web address: http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <math.h>
14 #include <assert.h>
15 #include <unistd.h>
16 #include <time.h>
17 #include <sys/time.h>
18
19 #include <starpu.h>
20
21 #define FPRINTF(ofile, fmt, ...) do { if (!getenv("STARPU_SSILENT")) {fprintf(
       ofile, fmt, ## __VA_ARGS__); }} while(0)
22
23 // measure time:
24 struct timeval start, end;
25
26 extern void mvt_cpu(void *descr[], void *_args);
27 extern void mvt_cuda(void *descr[], void *_args);
```

```
28
29 static struct starpu_perfmodel perf_model = {
30   .type = STARPU_HISTORY_BASED,
31   .symbol = "mvt",
32 };
33
34 static struct starpu_codelet cl =
35 {
36   /*CPU implementation of the codelet */
37   .cpu_funcs = { mvt_cpu },
38   .cpu_funcs_name = { "mvt_cpu" },
39   #ifdef STARPU_USE_CUDA
40   /* CUDA implementation of the codelet */
41   .cuda_funcs = { mvt_cuda },
42   #endif
43   .nbuffers = 5,
44   .modes = { STARPU_R, STARPU_RW, STARPU_RW, STARPU_R, STARPU_R },
45   .model = &perf_model
46 };
47
48 void init_array(int n, float *A, float *x1, float *x2, float *y1, float *y2)
49 {
50   int i, j;
51
52   for (i = 0; i < n; i++){
53     x1[i] = ((float) i) / n;
54     x2[i] = ((float) i + 1) / n;
55     y1[i] = ((float) i + 3) / n;
56     y2[i] = ((float) i + 4) / n;
57     for (j = 0; j < n; j++){
58       A[(i*n)+j] = ((float) i*j) / n;
59     }
60   }
61 }
62
63 void print_array(int n, float *x1, float *x2)
64 {
65   int i;
66
67   for (i = 0; i < n; i++) {
68     fprintf (stderr, "%0.2lf ", x1[i]);
69     fprintf (stderr, "%0.2lf ", x2[i]);
70     if (i % 20 == 0) fprintf (stderr, "\n");
71   }
72 }
73
74
75 int main(int argc, char *argv[])
76 {
77   /* Retrieve problem size */
78   if(argc < 1){
79     printf("Usage: %s problem_size debug\n",argv[0]);
80     exit(1);
```

```
81  }
82  int n = atoi(argv[1]);
83
84  float *A = (float *)malloc(n*n*sizeof(float));
85  float *x1 = (float *)malloc(n*sizeof(float));
86  float *x2 = (float *)malloc(n*sizeof(float));
87  float *y1 = (float *)malloc(n*sizeof(float));
88  float *y2 = (float *)malloc(n*sizeof(float));
89
90  // check if there's enough space for the allocation
91  if(!A){
92    printf("Allocation error for a - aborting.\n");
93    exit(1);
94  }
95
96  if(!x1){
97    printf("Allocation error for x1 - aborting.\n");
98    exit(1);
99  }
100
101 if(!x2){
102   printf("Allocation error for x2 - aborting.\n");
103   exit(1);
104 }
105
106 if(!y1){
107   printf("Allocation error for y1 - aborting.\n");
108   exit(1);
109 }
110
111 if(!y2){
112   printf("Allocation error for y2 - aborting.\n");
113   exit(1);
114 }
115
116 //initialization of the arrays
117 init_array(n, A, x1, x2, y1, y2);
118
119 /* Initialize StarPU with default configuration */
120 int ret = starpu_init(NULL);
121 if (ret == -ENODEV) goto enodev;
122
123 /* initialize performance model */
124 starpu_perfmodel_init(&perf_model);
125
126 starpu_data_handle_t matrixa_handle, vectorx1_handle, vectorx2_handle,
     vectory1_handle, vectory2_handle;
127 starpu_matrix_data_register(&matrixa_handle, STARPU_MAIN_RAM, (uintptr_t)A,
     n, n, n, sizeof(A[0]));
128 starpu_vector_data_register(&vectorx1_handle, STARPU_MAIN_RAM, (uintptr_t)x1
     , n, sizeof(x1[0]));
129 starpu_vector_data_register(&vectorx2_handle, STARPU_MAIN_RAM, (uintptr_t)x2
     , n, sizeof(x2[0]));
```

```
130   starpu_vector_data_register(&vectory1_handle, STARPU_MAIN_RAM, (uintptr_t)y1
        , n, sizeof(y1[0]));
131   starpu_vector_data_register(&vectory2_handle, STARPU_MAIN_RAM, (uintptr_t)y2
        , n, sizeof(y2[0]));
132
133   gettimeofday(&start, NULL);
134
135   starpu_task_insert( &cl,
136             STARPU_R, matrixa_handle,
137             STARPU_RW, vectorx1_handle,
138             STARPU_RW, vectorx2_handle,
139             STARPU_R, vectory1_handle,
140             STARPU_R, vectory2_handle,
141             0);
142
143   /* StarPU does not need to manipulate the vectors anymore so we can stop
        monitoring them */
144   starpu_data_unregister(matrixa_handle);
145   starpu_data_unregister(vectorx1_handle);
146   starpu_data_unregister(vectorx2_handle);
147   starpu_data_unregister(vectory1_handle);
148   starpu_data_unregister(vectory2_handle);
149
150   gettimeofday(&end, NULL);
151
152   /* terminate StarPU */
153   starpu_shutdown();
154
155 //  print_array(n, x1, x2);
156
157   free(A);
158   free(x1);
159   free(x2);
160   free(y1);
161   free(y2);
162
163   // calculate the time required for the calculation
164   double elapsed = (end.tv_sec*1000000+end.tv_usec)-(start.tv_sec*1000000+
        start.tv_usec);
165   fprintf(stderr, "\nElapsed time in application is: %lf msec\n", (elapsed)
        /1000.0F);
166
167   enodev:
168     return 77;
169 }
```

Code A.11: Matrix vector prodict and transpose CPU kernel implementation

```
1 #include <starpu.h>
2
3 void mvt_cpu(void *descr[], void *_args)
4 {
5   printf("cpu function\n");
6
```

```
  /* length of the matrix */
  unsigned n = STARPU_MATRIX_GET_NX(descr[0]);

  /* local copy of the matrix and vector pointers */
  float *A = (float *)STARPU_MATRIX_GET_PTR(descr[0]);
  float *x1 = (float *)STARPU_VECTOR_GET_PTR(descr[1]);
  float *x2 = (float *)STARPU_VECTOR_GET_PTR(descr[2]);
  float *y1 = (float *)STARPU_VECTOR_GET_PTR(descr[3]);
  float *y2 = (float *)STARPU_VECTOR_GET_PTR(descr[4]);

  int i, j;

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      x1[i] = x1[i] + A[(i*n)+j] * y1[j];
        }
    }

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      x2[i] = x2[i] + A[(j*n)+i] * y2[j];
    }
  }
}
```

Code A.12: Matrix vector prodict and transpose GPU kernel implementation

```
#include <starpu.h>

/* Thread block dimensions */
#define DIM_THREAD_BLOCK_X 32
#define DIM_THREAD_BLOCK_Y 8

__global__ void mvt_kernel1(int n, float *a, float *x1, float *y1)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < n) {
    int j;
    for(j=0; j < n; j++) {
      x1[i] += a[i * n + j] * y1[j];
    }
  }
  __syncthreads();
}

__global__ void mvt_kernel2(int n, float *a, float *x2, float *y2)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < n) {
    int j;
    for(j=0; j < n; j++) {
      x2[i] += a[j * n + i] * y2[j];
```

```
    }
  }
  __syncthreads();
}

extern "C" void mvt_cuda(void *descr[], void *_args)
{
  printf("cuda function\n");

  /* length of the matrix */
  unsigned n = STARPU_MATRIX_GET_NX(descr[0]);

  /* local copy of the matrix pointer */
  float *A = (float *)STARPU_MATRIX_GET_PTR(descr[0]);
  float *x1 = (float *)STARPU_VECTOR_GET_PTR(descr[1]);
  float *x2 = (float *)STARPU_VECTOR_GET_PTR(descr[2]);
  float *y1 = (float *)STARPU_VECTOR_GET_PTR(descr[3]);
  float *y2 = (float *)STARPU_VECTOR_GET_PTR(descr[4]);

  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
  dim3 grid((size_t)ceil((float)n/ ((float)DIM_THREAD_BLOCK_X)), 1);

  mvt_kernel1<<<grid,block>>>(n, A, x1 ,y1);

  mvt_kernel2<<<grid,block>>>(n, A, x2, y2);

  cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Code A.13: Symmetric rank-2K operations main code

```
/**
 * syr2k.cu: This file is part of the PolyBench/GPU 1.0 test suite.
 *
 *
 * Contact: Scott Grauer-Gray <sgrauerg@gmail.com>
 * Will Killian <killian@udel.edu>
 * Louis-Noel Pouchet <pouchet@cse.ohio-state.edu>
 * Web address: http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

#include <starpu.h>

#define FPRINTF(ofile, fmt, ...) do { if (!getenv("STARPU_SSILENT")) {fprintf(
    ofile, fmt, ## __VA_ARGS__); }} while(0)

```

```
23 // measure time:
24 struct timeval start, end;
25
26 extern void syr2k_cpu(void *descr[], void *_args);
27 extern void syr2k_cuda(void *descr[], void *_args);
28
29 static struct starpu_perfmodel perf_model = {
30   .type = STARPU_HISTORY_BASED,
31   .symbol = "syr2k",
32 };
33
34 static struct starpu_codelet cl =
35 {
36   /*CPU implementation of the codelet */
37   .cpu_funcs = { syr2k_cpu },
38   .cpu_funcs_name = { "syr2k_cpu" },
39   #ifdef STARPU_USE_CUDA
40   /* CUDA implementation of the codelet */
41   .cuda_funcs = { syr2k_cuda },
42   #endif
43   .nbuffers = 5,
44   .modes = { STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_RW },
45   .model = &perf_model
46 };
47
48 void init_arrays(int ni, int nj,
49     float *alpha,
50     float *beta,
51     float *A,
52     float *B,
53     float *C)
54 {
55   int i, j;
56
57   *alpha = 32412;
58   *beta = 2123;
59   for (i = 0; i < ni; i++) {
60     for (j = 0; j < nj; j++) {
61       A[(i*nj)+j] = ((float) i*j) / ni;
62       B[(i*nj)+j] = ((float) i*j) / ni;
63     }
64   }
65   for (i = 0; i < ni; i++) {
66     for (j = 0; j < ni; j++) {
67       C[(i*ni)+j] = ((float) i*j) / ni;
68     }
69   }
70 }
71
72 void print_array(int ni, float *C)
73 {
74   int i, j;
75
```

```
  for (i = 0; i < ni; i++)
    for (j = 0; j < ni; j++) {
  fprintf (stderr, "%0.2lf ", C[(i*ni)+j]);
  if ((i * ni + j) % 20 == 0) fprintf (stderr, "\n");
    }
  fprintf (stderr, "\n");
}


int main(int argc, char *argv[])
{
  /* Retrieve problem size. */
  if(argc < 2){
    printf("Usage: %s ni nj\n",argv[0]);
    exit(1);
  }
  int ni = atoi(argv[1]);
  int nj = atoi(argv[2]);

  /* Variable declaration/allocation. */
  float alpha;
  float beta;
  float *A = (float *)malloc(ni*nj*sizeof(float));
  float *B = (float *)malloc(ni*nj*sizeof(float));
  float *C = (float *)malloc(ni*ni*sizeof(float));

  // check if there's enough space for the allocation
  if(!A){
    printf("Allocation error for A - aborting.\n");
    exit(1);
  }

  if(!B){
    printf("Allocation error for B - aborting.\n");
    exit(1);
  }

  if(!C){
    printf("Allocation error for C - aborting.\n");
    exit(1);
  }

  //initialization of the arrays
  init_arrays(ni, nj, &alpha, &beta, A, B, C);

  /* Initialize StarPU with default configuration */
  int ret = starpu_init(NULL);
  if (ret == -ENODEV) goto enodev;

  /* initialize performance model */
  starpu_perfmodel_init(&perf_model);

  starpu_data_handle_t alpha_handle, beta_handle, matrixa_handle,
```

```
    matrixb_handle, matrixc_handle;
129   starpu_variable_data_register(&alpha_handle, STARPU_MAIN_RAM, (uintptr_t)&
      alpha, sizeof(float));
130   starpu_variable_data_register(&beta_handle, STARPU_MAIN_RAM, (uintptr_t)&
      beta, sizeof(float));
131   starpu_matrix_data_register(&matrixa_handle, STARPU_MAIN_RAM, (uintptr_t)A,
      ni, ni, nj, sizeof(A[0]));
132   starpu_matrix_data_register(&matrixb_handle, STARPU_MAIN_RAM, (uintptr_t)B,
      ni, ni, nj, sizeof(B[0]));
133   starpu_matrix_data_register(&matrixc_handle, STARPU_MAIN_RAM, (uintptr_t)C,
      ni, ni, ni, sizeof(C[0]));
134
135   gettimeofday(&start, NULL);
136
137   starpu_task_insert( &cl,
138         STARPU_R, alpha_handle,
139         STARPU_R, beta_handle,
140         STARPU_R, matrixa_handle,
141         STARPU_R, matrixb_handle,
142         STARPU_RW, matrixc_handle,
143         0);
144
145   /* StarPU does not need to manipulate the vectors anymore so we can stop
      monitoring them */
146   starpu_data_unregister(alpha_handle);
147   starpu_data_unregister(beta_handle);
148   starpu_data_unregister(matrixa_handle);
149   starpu_data_unregister(matrixb_handle);
150   starpu_data_unregister(matrixc_handle);
151
152   gettimeofday(&end, NULL);
153
154   /* terminate StarPU */
155   starpu_shutdown();
156
157 //  print_array(ni, C);
158
159   free(A);
160   free(B);
161   free(C);
162
163   // calculate the time required for the calculation
164   double elapsed = (end.tv_sec*1000000+end.tv_usec)-(start.tv_sec*1000000+
      start.tv_usec);
165   fprintf(stderr, "Elapsed time in application is: %lf msec\n", (elapsed)
      /1000.0F);
166
167   enodev:
168     return 77;
169 }
```

Code A.14: Symmetric rank-2K operations CPU kernel implementation

```
1 #include <starpu.h>
```

```
2
3  void syr2k_cpu(void *descr[], void *_args)
4  {
5    printf("cpu function\n");
6
7    /* length of the matrix */
8    unsigned ni = STARPU_MATRIX_GET_NX(descr[3]);
9    unsigned nj = STARPU_MATRIX_GET_NY(descr[3]);
10
11   /* local copy of the matrix pointers */
12   float *alpha = (float *)STARPU_VARIABLE_GET_PTR(descr[0]);
13   float *beta = (float *)STARPU_VARIABLE_GET_PTR(descr[1]);
14   float *A = (float *)STARPU_MATRIX_GET_PTR(descr[2]);
15   float *B = (float *)STARPU_MATRIX_GET_PTR(descr[3]);
16   float *C = (float *)STARPU_MATRIX_GET_PTR(descr[4]);
17
18   int i, j, k;
19
20   /*    C := alpha*A*B' + alpha*B*A' + beta*C */
21   for (i = 0; i < ni; i++) {
22     for (j = 0; j < ni; j++) {
23       C[(i*ni)+j] *= *beta;
24     }
25   }
26
27   for (i = 0; i < ni; i++) {
28     for (j = 0; j < ni; j++) {
29       for (k = 0; k < nj; k++) {
30         C[(i*ni)+j] += *alpha * A[(i*nj)+k] * B[(j*nj)+k];
31         C[(i*ni)+j] += *alpha * B[(i*nj)+k] * A[(j*nj)+k];
32       }
33     }
34   }
35 }
```

Code A.15: Symmetric rank-2K operations GPU kernel implementation

```
1  #include <starpu.h>
2
3  /* Thread block dimensions */
4  #define DIM_THREAD_BLOCK_X 32
5  #define DIM_THREAD_BLOCK_Y 8
6
7  __global__ void syr2k_kernel(int ni, int nj, float *alpha, float *beta, float
       *a, float *b, float *c)
8  {
9    int j = blockIdx.x * blockDim.x + threadIdx.x;
10   int i = blockIdx.y * blockDim.y + threadIdx.y;
11
12   if ((i < ni) && (j < nj)) {
13     c[i * ni + j] *= *beta;
14
15     int k;
16     for(k = 0; k < nj; k++) {
```

```
      c[i * ni + j] += *alpha * a[i * nj + k] * b[j * nj + k] + *alpha * b[i * nj + k] * a[j * nj + k];
    }
  }
  __syncthreads();
}

extern "C" void syr2k_cuda(void *descr[], void *_args)
{
  printf("cuda function\n");

  /* length of the matrix */
        unsigned ni = STARPU_MATRIX_GET_NX(descr[3]);
        unsigned nj = STARPU_MATRIX_GET_NY(descr[3]);

        /* local copy of the matrix pointers */
        float *alpha = (float *)STARPU_VARIABLE_GET_PTR(descr[0]);
        float *beta = (float *)STARPU_VARIABLE_GET_PTR(descr[1]);
        float *A = (float *)STARPU_MATRIX_GET_PTR(descr[2]);
        float *B = (float *)STARPU_MATRIX_GET_PTR(descr[3]);
        float *C = (float *)STARPU_MATRIX_GET_PTR(descr[4]);

  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
  dim3 grid((size_t)ceil((float)ni/ ((float)DIM_THREAD_BLOCK_X)), (size_t)( ceil( ((float)ni) / ((float)DIM_THREAD_BLOCK_Y) )));

  syr2k_kernel<<<grid,block>>>(ni, nj, alpha, beta, A, B, C);

  cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Code A.16: Brain modeling main code

```
#include <skepu>

#include "exa2pro.h"
#include <iostream>
#include <cstdlib>
#include <chrono>

#include "defines.h"

//[[skepu::userfunction]]
inline float Exp(float x){
  return exp(x); // do not use expf or the compiler crashes, giving no information why !
}

// gap-junction current is a non-linear function of voltage, see Schweighofer 2004 gap unction model
//[[skepu::userfunction]]
inline float fGap(float w, float vi, float vj){
  float v = (vj - vi);
  return -v * w * Exp(v * v * (-1.0/100.0)) ;
```

```
20 }
21
22 // T.T.Wang's int -> int hash function: http://www.concentric.net/~Ttwang/tech
       /inthash.htm
23 inline uint64_t hash64shift(uint64_t key){
24   key = (~key) + (key << 21); // key = (key << 21) - key - 1;
25   key = key ^ (key >> 24);
26   key = (key + (key << 3)) + (key << 8); // key * 265
27   key = key ^ (key >> 14);
28   key = (key + (key << 2)) + (key << 4); // key * 21
29   key = key ^ (key >> 28);
30   key = key + (key << 31);
31   return key;
32 }
33
34 inline double Random_Uniform( double from, double to, uint64_t seed){
35   const uint32_t sample_scale = (1 << 24); // how many bits wide should a
       random sample be?
36   uint64_t key = seed;
37   int64_t sample = int64_t(hash64shift(key) % sample_scale);
38   return from + (from - to) * float(sample / sample_scale);
39 }
40
41 /* SkePU incomplete type fix. */
42 auto
43 operator==(State const & s1, State const & s2)
44 -> bool
45 {
46   return s1.v == s2.v
47     && s1.m == s2.m
48     && s1.h == s2.h
49     && s1.n == s2.n;
50 }
51
52 auto
53 operator!=(State const & s1, State const & s2)
54 -> bool
55 {
56   return !(s1 == s2);
57 }
58 /* End of SkePU incomplete type fix. */
59
60 // Immediately parallel initialization of weight matrix
61 //[[skepu::userfunction]]
62 inline float initWeight(int i, int j, const float wConstant, const float
       pDensity, const uint64_t seed){
63   //return 0.01;
64
65   // for symmetry
66   if( i > j ){
67     auto t = i;
68     i = j;
69     j = t;
```

```
70    }
71
72    const uint32_t sample_scale = (1 << 24); // how many bits wide should a
       random sample be?
73    const uint32_t fraction_of_scale = uint32_t( pDensity * sample_scale ); //
       density scaled to this bit width
74    uint64_t key = ( (uint64_t(i) << 32) + uint64_t(j) ) ^ 0xA6DEE8D2E4D2DEE6ULL
       ^ seed;
75    uint64_t sample = hash64shift(key) % sample_scale;
76
77    float val = 0;
78    if(i == j) val = 0; //no self connnections
79    else if(sample < fraction_of_scale){
80      val = wConstant;
81    }
82
83    return val;
84  }
85
86  // Integrate results for next timestep
87  State InnerDynamics_Integrate( const Constants parms, const State statePrev,
       float iGapTotal, float time, float dt ){
88
89    // state at last timestep
90    const float V = statePrev.v;
91    const float m = statePrev.m;
92    const float h = statePrev.h;
93    const float n = statePrev.n;
94    // constants at last timestep
95    const float C     = parms.C    ;
96    const float tOn   = parms.tOn  ;
97    const float tOff  = parms.tOff ;
98    const float iStim = parms.iStim;
99
100   // caclulate inner dynamics
101   const float gLeak  =     0.3;
102   const float gNabar =   120.0;
103   const float gKbar  =    36.0;
104   const float eLeak  =    10.6;
105   const float eNa    =   115.0;
106   const float eK     = -  12.0;
107
108   // get currents from ion channels
109   float gNa = gNabar * m * m * m * h;
110   float gK  = gKbar  * n * n * n * n;
111
112   float iLeak = (eLeak - V) * gLeak;
113   float iNa = (eNa - V) * gNa;
114   float iK  = (eK  - V) * gK ;
115   float iApp  = 0; // applied current
116   if( tOn <= time && time <= tOff ){
117     iApp = iStim;
118   }
```

```
119
120   float iTotal = iGapTotal + iNa + iK + iLeak + iApp; // inward current
121
122   // get dynamics of ion channels
123   float m_alpha = (0.1 * (25 - V) ) / (Exp((25 - V) / 10) - 1) ;
124   float m_beta = 4 * Exp( (-V) / 18 );
125
126   float h_alpha = 0.07 * Exp( (-V) / 20 );
127   float h_beta = 1 / (Exp( (30 - V) / 10 ) + 1) ;
128
129   float n_alpha = (0.01 * (10 - V) ) / (Exp((10 - V) / 10) - 1) ;
130   float n_beta = 0.125 * Exp( (-V) / 80 );
131
132   // integrate results for next timestep
133   State stateNext;
134   stateNext.v = statePrev.v + dt * ( iTotal ) / parms.C;
135
136   stateNext.m = statePrev.m + dt * ( m_alpha * (1 - m) - m_beta * m );
137   stateNext.h = statePrev.h + dt * ( h_alpha * (1 - h) - h_beta * h );
138   stateNext.n = statePrev.n + dt * ( n_alpha * (1 - n) - n_beta * n );
139
140   return stateNext;
141 }
142
143 // Initialize the state variables of each neuron
144 //[[skepu::userfunction]]
145 State initState_Skepu(uint64_t random_seed){
146   random_seed ^= 0x20000000ULL;
147   State ret;
148   ret.v = Random_Uniform(0,2,random_seed);
149   ret.m = 0.5;
150   ret.h = 0.5;
151   ret.n = 0.5;
152   return ret;
153 };
154 // Initialize the constants of each neuron
155 //[[skepu::userfunction]]
156 Constants initConsts_Skepu(skepu::Index1D ind, uint64_t random_seed){
157   size_t i = ind.i;
158
159   Constants ret;
160   ret.C     = 1.000;
161   ret.tOn   = 0.010; // msec
162   ret.tOff  = 0.100; // msec
163   ret.iStim = 0.300;
164   if( i % 10 == 0 ){
165     ret.iStim = 0.300;
166   }
167   else{
168     ret.iStim = 0.000;
169   }
170
171   return ret;
```

```
172 }
173 // Initialize the weight matrix of neuron connectivity
174 //[[skepu::userfunction]]
175 float initMatrix_Skepu(skepu::Index2D rowcol, const float wConstant, const
        float pDensity, const uint64_t seed){
176   return initWeight(rowcol.row, rowcol.col, wConstant, pDensity, seed);
177 }
178
179 // Process each neuron wih this Map-ped function
180 //[[skepu::userfunction]]
181 State SimulateNeuron_Skepu(skepu::Index1D row, const skepu::Mat<float> m,
        const skepu::Vec<State> state, const skepu::Vec<Constants> constants,
        float time, float dt){
182
183   // get context for i-th neuron
184   auto i = row.i;
185   Constants parms = constants[i];
186   State statePrev = state[i];
187
188   // get currents from gap junctions
189   float iGapTotal = 0;
190   int rows = m.rows;
191   int cols = m.cols;
192   for (size_t j = 0; j < cols; j++){
193     State sti = state.data[i];
194     State stj = state.data[j];
195     iGapTotal += fGap( m.data[i*cols+j] , sti.v, stj.v ); // m.data[row.i * m.
        cols + j] * *(v.data[i] - v.data[j]);
196   }
197
198   // get next state
199   State stateNext = InnerDynamics_Integrate(parms, statePrev, iGapTotal, time,
        dt);
200
201   return stateNext;
202 }
203
204 int main(int argc, char *argv[]){
205
206   bool debug = false;
207   bool write_to_file = false;
208
209   // Fixed model parameters
210   const uint64_t random_seed = ~0xAC949A939A8B9092ULL; // whatever
211   float dt = 0.010; // msec
212
213   const float connWeight = 0.01; // Connectivity matrix configuration
214
215   if (argc < 6 || argc > 6){
216     skepu::external([&]{
217       printf("Usage: %s neurons steps density file backend\n", argv[0]);});
218     exit(1);
219   }
```

```
220
221   // Simulation parameters
222   size_t neurons = atoi(argv[1]); // zero in case of error
223   int steps = atoi(argv[2]); // zero in case of error
224   float connDensity = 0.12;
225   if(!( sscanf( argv[3], "%f", &connDensity) == 1 )){
226     connDensity = NAN; // will be caught later
227   }
228
229   const char *outfile = argv[4];
230   if( strcmp(outfile, "NIL") == 0 ){
231     write_to_file = false;
232   }
233   else if ( strcmp(outfile, "DBGNIL") == 0 ){
234     debug = true;
235     write_to_file = false;
236   }
237   else if ( strcmp(outfile, "DBGTXT") == 0 ){
238     debug = true;
239     write_to_file = true;
240     outfile = "debug.txt";
241   }
242   else{
243     write_to_file = true;
244   }
245
246   // setup the SkePU parallelized skeleton functors
247   auto spec = skepu::BackendSpec{skepu::Backend::typeFromString(argv[5])};
248   auto initit_S = skepu::Map<0>(initState_Skepu);
249   initit_S.setBackend(spec);
250   auto initit_C = skepu::Map<0>(initConsts_Skepu);
251   initit_C.setBackend(spec);
252   auto initit_W = skepu::Map<0>(initMatrix_Skepu);
253   initit_W.setBackend(spec);
254
255   auto SimulateTimestepPerNeuron = skepu::Map<0>(SimulateNeuron_Skepu);
256   SimulateTimestepPerNeuron.setBackend(spec);
257
258   // check parameters
259   bool parm_fail = false;
260
261   if(!( steps > 0 )){
262     skepu::external([&]{
263       printf("Steps should be a positive integer\n");});
264     parm_fail = true;
265   }
266   if(!( neurons > 0 )){
267     skepu::external([&]{
268       printf("Neurons should be a positive integer\n");});
269     parm_fail = true;
270   }
271   if(!( 0 <= connDensity && connDensity <= 1 )){
272     skepu::external([&]{
```

```
273       printf("Density should be between zero(none) and one(full)\n");});
274     parm_fail = true;
275   }
276
277   if(parm_fail){
278     skepu::external([&]{
279       printf("Cannot run simulation due to wrong parameters\n");});
280     exit(5);
281   }
282
283   EXA2PRO_INITIALIZE();
284
285   // the data sets of the problem
286   skepu::Vector<State> state(neurons), state2(neurons);
287   skepu::Vector<State> returned(neurons), returned2(neurons); // 'returned'
       ones are for mapping
288   skepu::Vector<Constants> constants(neurons);
289   skepu::Matrix<float> weight_matrix(neurons, neurons);
290
291   // initialize constants & state
292
293   initit_S(state, random_seed);
294   initit_C(constants, random_seed);
295   initit_W(weight_matrix, connWeight, connDensity, random_seed);
296
297   // debug print helpers
298   auto printit = [neurons](/*const*/ skepu::Vector<State> &state){
299     for (size_t i = 0; i < neurons; i++){
300       printf("%f ",state(i).v);
301     }
302     printf("\n");
303   };
304   if(debug){
305     skepu::external(skepu::read(weight_matrix, state), [&]{
306       printf("Weight Matrix: \n");
307       for(size_t i = 0; i < neurons ; i++){
308         for(size_t j = 0; j < neurons ; j++){
309           printf("%.0f ", float(weight_matrix(i,j))*100 );
310         }
311         printf("\n");
312       }
313       printf("Initial Voltages: \n");
314       printit(state);
315     });
316   }
317
318   // initialize logging
319   FILE *fout = NULL;
320   if( write_to_file ){
321     skepu::external([&]{
322       fout = fopen(outfile, "w"); // text for now
323       if(!fout){
324         perror("opening text output file");
```

```
325         exit(1);
326       }
327       fprintf( fout, "Simulation %d %f %zd %f \n", steps, dt, neurons,
    connDensity );
328       fprintf( fout, "Time\t" );
329       for( size_t i = 0; i < neurons; i++ ){
330         fprintf( fout, "V[%zd]\t",i );
331       }
332       fprintf( fout, "\n" );
333     });
334   }
335
336   if(debug){
337     skepu::external([&]{
338       printf("SIMULATION START: \n");});
339   }
340
341   // for buffer swapping. Use pointers instead of references, to better work
     with containers
342   auto *pOne = &state;
343   auto *pTwo = &state2;
344
345   auto start = std::chrono::steady_clock::now();
346   // run the simulation
347   for(int t = 0; t < steps; t++){
348     float time = t * dt; // time point in simulation
349
350     directSim(returned2, weight_matrix, *pOne, constants, time, dt);
351     auto end = std::chrono::steady_clock::now();
352
353     if(debug){
354       skepu::external(skepu::read(returned, returned2, weight_matrix, *pOne),
    [&]{
355         printit(returned);
356
357         SimulateTimestepPerNeuron (returned, weight_matrix, *pOne, constants,
    time, dt);
358         printit(returned);
359         for (size_t i = 0; i < neurons; i++){
360           State st1 = returned (i);
361           State st2 = returned2(i);
362           if (st1.v != st2.v){
363             printf("Output error at index %zd: %f should be %f\n", i, st1.v,
    st2.v);
364           }
365         }
366       });
367     }
368
369     *pTwo = returned;
370
371     //file output
372     if( write_to_file ){
```

```
      skepu::external(skepu::read(*pTwo), [&]{
        // simulation takes more thime than output, just print ASCII for now
        fprintf( fout, "%.6f\t", time);
        for( size_t i = 0; i < neurons; i++ ){
          fprintf( fout, "%+03.6f\t", State((*pTwo)(i)).v );
        }
        fprintf( fout, "\n" );
      });
    }

    std::swap(pOne, pTwo); // swap the buffers, ready for new iteration
  }

  std::cout << "Elapsed time is " << chrono::duration_cast<chrono::seconds>(
    end - start).count() << " sec\n";

  // done!
  if( write_to_file ){
    skepu::external([&]{
    fclose( fout );});
  }

  EXA2PRO_SHUTDOWN();
  return 0;
}
```

Code A.17: Brain modeling define functions

```
#ifndef DEFINES_H
#define DEFINES_H

#include <algorithm>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

//[[skepu::userfunction]]
extern float Exp(float /*x*/);

// gap-junction current is a non-linear function of voltage, see Schweighofer
    2004 gap unction model
//[[skepu::userfunction]]
extern float fGap(float, float, float);

// constant parameters for each neuron, assume a classical HH neuron for this
    miniapp
struct Constants{
  float C;         // compartment capacitance
  float iStim, tOn, tOff; // stimulus DC pulse intensity, start and end
};
// state variables for each neuron, assume a classical HH neuron for this
    miniapp
struct State{
```

```
  float v;      // classic HH compartment voltage
  float m,h,n; // classic HH gate activation variables
};

extern State InnerDynamics_Integrate( const Constants, const State, float,
    float, float );

#endif
```

Code A.18: Brain modeling xml descriptor

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
  xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1
   ComponentMetaData0.1.xsd">

  <x2p:implementation name="main" providedInterface="main" targetPlatform="CPU
   " >
    <x2p:requiredInterfaces>
      <x2p:requiredInterface name="directSim" />
    </x2p:requiredInterfaces>

    <x2p:sourceFiles>
      <x2p:sourceFile version="1.0" language="C++">
        <x2p:compilation type="link" compiler="g++" version="4.5" flags="$(
   LDFLAGS) $(shell pkg-config --libs cuda-9.2 cudart-9.2)" output="main" />
      </x2p:sourceFile>

      <x2p:sourceFile name="main.cpp" version="1.0" language="C++">
        <x2p:compilation compiler="g++" version="4.5" flags="-I ~/git/skepu/
   skepu-headers/src -std=c++11"/>
      </x2p:sourceFile>
    </x2p:sourceFiles>
  </x2p:implementation>
</x2p:component>
```

Code A.19: Brain modeling parameter xml descriptor

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
               xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1
   ComponentMetaData0.1.xsd">

    <x2p:interface name="directSim">
        <x2p:parameters>
            <x2p:parameter name="res" type="skepu::Vector" elemType="State"
   accessMode="write" />
            <x2p:parameter name="m" type="skepu::Matrix" elemType="float"
   accessMode="read" />
            <x2p:parameter name="state" type="skepu::Vector" elemType="State"
   accessMode="read" />
            <x2p:parameter name="constants" type="skepu::Vector" elemType="
   Constants" accessMode="read" />
```

```
            <x2p:parameter name="time" type="float" accessMode="read" />
            <x2p:parameter name="dt" type="float" accessMode="read" />
        </x2p:parameters>
    </x2p:interface>

</x2p:component>
```

Code A.20: Brain modeling CPU kernel implementation

```
#include "skepu"

#include "../../defines.h"

void directSim_cpu(skepu::Vector<State> &res, skepu::Matrix<float> &m, skepu::
    Vector<State> &state, /*const*/ skepu::Vector<Constants> &constants, float
     time, float dt) {
  int rows = m.size_i();
  int cols = m.size_j();

  for (int i = 0; i < rows; ++i){

    // get context for i-th neuron
    Constants parms = constants(i);
    State statePrev = state(i);

    // get currents from gap junctions
    float iGapTotal = 0;
    for (int j = 0; j < cols; ++j){
      State sti = state(i);
      State stj = state(j);
      iGapTotal += fGap(m(i, j) , sti.v, stj.v); // fGap( m[i*cols+j] , sti.v,
    stj.v );
    }

    // get next state
    State stateNext = InnerDynamics_Integrate(parms, statePrev, iGapTotal,
    time, dt);

    res(i) = stateNext;
  }
}
```

Code A.21: Brain modeling CPU kernel descriptor

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
  xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1
    ComponentMetaData0.1.xsd">

  <x2p:implementation name="directSim_cpu" targetPlatform="CPU"
    providedInterface="directSim">
    <x2p:sourceFiles>
      <x2p:sourceFile name="directSimcpu.cpp" version="1.0" language="C">
```

```
        <x2p:compilation compiler="g++" version="5.4" flags="-I ~/git/skepu/skepu-headers/src -std=c++11" />
      </x2p:sourceFile>
    </x2p:sourceFiles>
  </x2p:implementation>
</x2p:component>
```

Code A.22: Brain modeling GPU kernel implementation

```
#include "skepu"
#include <starpu.h>
#include "../../defines.h"

#define Block_Size 16

static __global__ void directSim_kernel(State *res, float *m, State *state, /*const*/ Constants *constants, float time, float dt, int rows, int cols)
{
    int i = blockIdx.y*blockDim.y+threadIdx.y;
    int j = blockIdx.x*blockDim.x+threadIdx.x;

    if (i < rows && j < cols){
        // get context for i-th neuron
  Constants parms = constants[i];
  State statePrev = state[i];

  // get currents from gap junctions
  float iGapTotal = 0;
  State sti = state[i];
  State stj = state[j];
  float dif = stj.v - sti.v;
  iGapTotal = -dif * m[i,j] * exp(dif * dif * (-1.0/100.0));

  const float V = statePrev.v;
  const float m = statePrev.m;
  const float h = statePrev.h;
  const float n = statePrev.n;
  // constants at last timestep
  const float C     = parms.C    ;
  const float tOn   = parms.tOn  ;
  const float tOff  = parms.tOff ;
  const float iStim = parms.iStim;

  // caclulate inner dynamics
  const float gLeak  =     0.3;
  const float gNabar =   120.0;
  const float gKbar  =    36.0;
  const float eLeak  =    10.6;
  const float eNa    =   115.0;
  const float eK     = -  12.0;

  // get currents from ion channels
  float gNa = gNabar * m * m * m * h;
  float gK  = gKbar  * n * n * n * n;
```

```
45
46    float iLeak = (eLeak - V) * gLeak;
47    float iNa = (eNa - V) * gNa;
48    float iK  = (eK  - V) * gK ;
49    float iApp  = 0; // applied current
50    if( tOn <= time && time <= tOff ){
51      iApp = iStim;
52    }
53
54    float iTotal = iGapTotal + iNa + iK + iLeak + iApp; // inward current
55
56    // get dynamics of ion channels
57    float m_alpha = (0.1 * (25 - V) ) / (exp((25 - V) / 10) - 1) ;
58    float m_beta = 4 * exp( (-V) / 18 );
59
60    float h_alpha = 0.07 * exp( (-V) / 20 );
61    float h_beta = 1 / (exp( (30 - V) / 10 ) + 1) ;
62
63    float n_alpha = (0.01 * (10 - V) ) / (exp((10 - V) / 10) - 1) ;
64    float n_beta = 0.125 * exp( (-V) / 80 );
65
66    // integrate results for next timestep
67    State stateNext;
68    stateNext.v = statePrev.v + dt * ( iTotal ) / parms.C;
69
70    stateNext.m = statePrev.m + dt * ( m_alpha * (1 - m) - m_beta * m );
71    stateNext.h = statePrev.h + dt * ( h_alpha * (1 - h) - h_beta * h );
72    stateNext.n = statePrev.n + dt * ( n_alpha * (1 - n) - n_beta * n );
73    res[i] = stateNext;
74      }
75
76  }
77
78  void directSim_cuda(skepu::Vector<State> &res, skepu::Matrix<float> &m, skepu
      ::Vector<State> &state, /*const*/ skepu::Vector<Constants> &constants,
      float time, float dt) {
79    int rows = m.size_i();
80    int cols = m.size_j();
81
82      unsigned int grid_rows = (rows + Block_Size - 1)/ Block_Size;
83      unsigned int grid_cols = (cols + Block_Size - 1)/ Block_Size;
84
85      dim3 dimGrid(grid_cols, grid_rows);        dim3 dimBlock(Block_Size,
      Block_Size);
86
87
88      directSim_kernel<<<dimGrid,dimBlock>>>( res.getAddress(), m.getAddress(),
      state.getAddress(), constants.getAddress(), time, dt, rows, cols);
89      cudaStreamSynchronize(starpu_cuda_get_local_stream());
90      cudaThreadSynchronize();
91
92  }
```

Code A.23: Brain modeling GPU kernel descriptor

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
  xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1
   ComponentMetaData0.1.xsd">

  <x2p:implementation name="directSim_cuda" targetPlatform="CUDA"
   providedInterface="directSim">
    <x2p:sourceFiles>
      <x2p:sourceFile name="directSimcuda.cu" version="1.0" language="CUDA">
        <x2p:compilation compiler="nvcc" flags="--compiler-options -
   fpermissive -I ~/git/skepu/skepu-headers/src -std=c++11" />
      </x2p:sourceFile>
    </x2p:sourceFiles>
  </x2p:implementation>
</x2p:component>
```

Code A.24: Brain modeling generated wrapper file

```
#ifndef DIRECTSIM_WRAPPERR
#define DIRECTSIM_WRAPPERR

typedef struct
{
  float time;float dt;
} ROA_directSim;

typedef struct
{
  struct starpu_codelet cl_directSim;

   int cl_directSim_init;
} struct_directSim;

// include statement

// extern statement
extern void directSim_cpu( skepu::Vector<State> &res, skepu::Matrix<float> &m
   , skepu::Vector<State> &state, skepu::Vector<Constants> &constants, float
   time, float dt);

void directSim_cpu_wrapper (void *buffers[], void *_args)
{
/* exa2pro container declarations, if any */
skepu::Vector<State> res_handle((State *)STARPU_VECTOR_GET_PTR( (struct
   starpu_vector_interface *)buffers[0]), STARPU_VECTOR_GET_NX( (
   struct_vector_interface *)buffers[0]), false );
skepu::Matrix<float> m_handle((float *)STARPU_MATRIX_GET_PTR( (struct
   starpu_matrix_interface *)buffers[1]), STARPU_MATRIX_GET_NX( (
   struct_matrix_interface *)buffers[0]), STARPU_MATRIX_GET_NY( (struct
   starpu_matrix_interface *)buffers[1]), false );
skepu::Vector<State> state_handle((State *)STARPU_VECTOR_GET_PTR( (struct
   starpu_vector_interface *)buffers[2]), STARPU_VECTOR_GET_NX( (
```

```
   struct_vector_interface *)buffers[2]), false );
skepu::Vector<Constants> constants_handle((Constants *)STARPU_VECTOR_GET_PTR(
   (struct starpu_vector_interface *)buffers[3]), STARPU_VECTOR_GET_NX( (
   struct_vector_interface *)buffers[3]), false );

directSim_cpu (
 res_handle,m_handle,state_handle,constants_handle,((ROA_directSim *)_args)->
   time,((ROA_directSim *)_args)->dt
);

}

extern void directSim_cuda( skepu::Vector<State> &res, skepu::Matrix<float> &
   m, skepu::Vector<State> &state, skepu::Vector<Constants> &constants, float
    time, float dt);

void directSim_cuda_wrapper (void *buffers[], void *_args)
{
/* exa2pro container declarations, if any */
skepu::Vector<State> res_handle((State *)STARPU_VECTOR_GET_PTR( (struct
   starpu_vector_interface *)buffers[0]), STARPU_VECTOR_GET_NX( (
   struct_vector_interface *)buffers[0]), false );
skepu::Matrix<float> m_handle((float *)STARPU_MATRIX_GET_PTR( (struct
   starpu_matrix_interface *)buffers[1]), STARPU_MATRIX_GET_NX( (
   struct_matrix_interface *)buffers[0]), STARPU_MATRIX_GET_NY( (struct
   starpu_matrix_interface *)buffers[1]), false );
skepu::Vector<State> state_handle((State *)STARPU_VECTOR_GET_PTR( (struct
   starpu_vector_interface *)buffers[2]), STARPU_VECTOR_GET_NX( (
   struct_vector_interface *)buffers[2]), false );
skepu::Vector<Constants> constants_handle((Constants *)STARPU_VECTOR_GET_PTR(
   (struct starpu_vector_interface *)buffers[3]), STARPU_VECTOR_GET_NX( (
   struct_vector_interface *)buffers[3]), false );

directSim_cuda (
 res_handle,m_handle,state_handle,constants_handle,((ROA_directSim *)_args)->
   time,((ROA_directSim *)_args)->dt
);

}

static struct_directSim * objSt_directSim = NULL;

void directSim (  skepu::Vector<State> &res, skepu::Matrix<float> &m, skepu::
   Vector<State> &state, skepu::Vector<Constants> &constants, float time,
   float dt )
{

if( objSt_directSim == NULL)
{
objSt_directSim = ( struct_directSim *) malloc (sizeof( struct_directSim ));

memset( &(objSt_directSim->cl_directSim), 0, sizeof(objSt_directSim->
   cl_directSim));
```

```
61
62  objSt_directSim->cl_directSim_init = 0;
63  }
64
65  if(! objSt_directSim->cl_directSim_init ) // codelete initialization only
      once, at first invocation
66  {
67  objSt_directSim->cl_directSim.where =0|STARPU_CPU|STARPU_CUDA;
68
69  objSt_directSim->cl_directSim.cpu_funcs[0]=directSim_cpu_wrapper;
70 objSt_directSim->cl_directSim.cpu_funcs[1]=NULL;
71
72  objSt_directSim->cl_directSim.cuda_funcs[0]=directSim_cuda_wrapper;
73 objSt_directSim->cl_directSim.cuda_funcs[1]=NULL;
74
75  objSt_directSim->cl_directSim.nbuffers = 4;
76
77   objSt_directSim->cl_directSim.modes[0] = STARPU_W;
78 objSt_directSim->cl_directSim.modes[1] = STARPU_R;
79 objSt_directSim->cl_directSim.modes[2] = STARPU_R;
80 objSt_directSim->cl_directSim.modes[3] = STARPU_R;
81
82  objSt_directSim->cl_directSim_init = 1;
83  }
84
85   starpu_data_handle_t res_handle;
86   starpu_vector_data_register(&res_handle, STARPU_MAIN_RAM, (uintptr_t)res.
      getAddress(), res.size(), sizeof(res[0]));
87 starpu_data_handle_t m_handle;
88 starpu_matrix_data_register(&m_handle, STARPU_MAIN_RAM, (uintptr_t)m.
      getAddress(), m.size_i(), m.size_i(), m.size_j(), sizeof(float));
89 starpu_data_handle_t state_handle;
90 starpu_vector_data_register(&state_handle, STARPU_MAIN_RAM, (uintptr_t)state.
      getAddress(), state.size(), sizeof(state[0]));
91 starpu_data_handle_t constants_handle;
92 starpu_vector_data_register(&constants_handle, STARPU_MAIN_RAM, (uintptr_t)
      constants.getAddress(), constants.size(), sizeof(constants[0]));
93
94  {
95  ROA_directSim *arg_directSim = (ROA_directSim *)malloc (sizeof (ROA_directSim
      ));
96
97   arg_directSim->time=time;arg_directSim->dt=dt;
98
99  struct starpu_task *task = starpu_task_create();
100
101 task->synchronous = 0;
102
103 task->cl = &(objSt_directSim->cl_directSim);
104
105   task->handles[0] = res_handle;
106 task->handles[1] = m_handle;
107 task->handles[2] = state_handle;
```

```
108 task->handles[3] = constants_handle;
109
110  task->callback_func = free;
111 task->callback_arg = (void *) arg_directSim;
112 task->cl_arg = arg_directSim;
113  task->cl_arg_size = sizeof(ROA_directSim);
114
115  if(x2p_nextcall_bitvector != 0){
116  // cout <<"next_bit_vector = "<< x2p_nextcall_bitvector << endl;
117  unsigned long int x2p_where = x2p_nextcall_bitvector & objSt_directSim->
       cl_directSim.where ;
118  // cout << "x2p_where = " << x2p_where <<endl;
119  task->where = x2p_where;
120  }
121
122  /* execute the task on any eligible computational ressource */
123  int ret = starpu_task_submit(task);
124
125  if (ret == -ENODEV)
126  {
127  fprintf(stderr, "ERROR: No worker may execute this task\n");
128  x2p_nextcall_bitvector = 0;
129  return;
130  }
131
132  }
133
134  starpu_data_unregister(res_handle);
135  starpu_data_unregister(m_handle);
136  starpu_data_unregister(state_handle);
137  starpu_data_unregister(constants_handle);
138
139  x2p_nextcall_bitvector = 0;
140
141  }
142
143  #endif
```

REFERENCES

[1] HPX documentation 1.3.0. https://hpx-docs.stellar-group.org/tags/1.3.0/pdf/HPX.pdf, 2019.

[2] OmpSs-2 specification. https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf, 2020.

[3] OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, 2020.

[4] StarPU Handbook for StarPU 1.3.7. https://files.inria.fr/starpu/doc/starpu.pdf, 2020.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par*, 2009.

[6] R. M. Badia, J. M. Pérez, E. Ayguadé, and J. Labarta. Impact of the memory hierarchy on shared memory architectures in multicore programming models. *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 437–445, 2009.

[7] R. Blumofe, C. F. Joerg, B. Kuszmaul, C. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distributed Comput.*, 37:55–69, 1996.

[8] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Transactions on Architecture and Code Optimization*, 12:41:1–41:22, 2016.

[9] U. Dastgeer, L. Li, and C. Kessler. The peppher composition tool: performance-aware composition for gpu-based systems. *Computing*, 96:1195–1211, 2013.

[10] A. J. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. Zapata. Evaluation of the task programming model in the parallelization of wavefront problems. *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 257–264, 2010.

[11] A. Ernstsson. *Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems*. 11 2020.

[12] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Task-based programming with OmpSs and its application. In *Euro-Par Workshops*, 2014.

[13] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx: A task based programming model in a global address space. In *PGAS*, 2014.

[14] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, page 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] C. Kessler and J. Keller. Models for parallel computing: Review and perspectives. *PARS-Mitteilungen, ISSN 0177-0454*, 24:13–29, 10 2007.

[16] C. Kessler, S. Zouzoula, J. Ahlqvist, A. Ernstsson, S. Memeti, O. Sysoev, T. Becker, A. Cramb, and N. Voss. D3.6 Final version of the composition and performance modelling framework, Submitted. 2020.

[17] C. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51:244–257, 2009.

[18] A. Leist and A. Gilman. A comparative analysis of parallel programming models for c. 2014.

[19] S. Panagiotou, A. Ernstsson, J. Ahlqvist, L. Papadopoulos, C. Kessler, and D. Soudris. Portable exploitation of parallel and heterogeneous hpc architectures in neural simulation using skepu. *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, pages 74–77, 2020.

[20] L. Papadopoulos, D. Soudris, I. Walulya, and P. Tsigas. Customization methodology for implementation of streaming aggregation in embedded systems. *Journal of Systems Architecture*, 66-67:48–60, 2016.

[21] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *The International Journal of High Performance Computing Applications*, 23:284 – 299, 2009.

[22] A. Podobas, M. Brorsson, and K.-F. Faxén. A comparison of some recent task-based parallel programming models. 2010.

[23] A. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15:66–71, 2013.

[24] D. Soudris, L. Papadopoulos, C. Kessler, D. D. Kehagias, A. Papadopoulos, P. Seferlis, A. Chatzigeorgiou, A. Ampatzoglou, S. Thibault, R. Namyst, D. Pleiter, G. Gaydadjiev, T. Becker, and M. Haefele. Exa2pro programming environment: architecture and applications. *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018.

[25] P. Stpiczynski. Language-based vectorization and parallelization using intrinsics, openmp, tbb and cilk plus. *The Journal of Supercomputing*, 74:1461–1472, 2017.

[26] S. Thibault. On runtime systems for task-based programming on heterogeneous platforms. 2018.

[27] G. Tzanos et.al. Applying StarPU Runtime System to Scientific Applications: Experiences and Lessons Learned. In *2nd International Workshop on Parallel Optimization using/for Multi- and Many-core High Performance Computing (POMCO)*, 2021.

[28] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB C++ Parallel Programming with Threading Building Blocks*. Apress, Berkeley, CA, 2019.