SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D4.7 – Advanced Version of the Compilation Tools

Due date of deliverable: 31st March 2014
Actual Submission: 19th May 2014

Start date of the project: January 1st, 2010 Duration: 51 months

Lead contractor for the deliverable: INRIA

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Author	Organization	Change History
1	Albert Cohen	INRIA	Template
2	Ian Watson	UNIMAN	Scala tool flow
3	Albert Cohen	INRIA	First complete version
4	Laurent Morin	CAPS	Fixes
5	Laurent Morin	CAPS	HMPP tool flow
6	Albert Cohen	INRIA	Corrections after review by Somnath Mazumdar (UNISI) Stéphane Zuckerman (UDEL)
7	Roberto Giorgi	UNISI	Review

Release Approval

Name	Role	Date
Albert Cohen	Originator	07/04/2014
Albert Cohen	WP Leader	07/04/2014
Roberto Giorgi	Project Coordinator for formal deliverable	10/05/2014

TABLE OF CONTENTS

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Albert Cohen, François Gindraud, Feng Li, Antoniu Pop
INRIA

Rosa Badia, Nacho Navarro, Tomasz Patejko
BSC

Ian Watson, Behram Khan and Mikel Lujan
UNIMAN

Roberto Giorgi, Andrea Mondelli
UNISI

Laurent Morin
CAPS

Printed in Siena, Italy, Europe.
Part number: *please refer to the File name in the document footer*.

1 Glossary

OpenMP – Parallel programming pragma language on top of C, C++ and FORTRAN. In this deliverable, we refer to the OpenMP specification version 3.1.

http://www.openmp.org

OpenStream – Dataflow streaming extension of OpenMP, for the C language, implemented as a patch to GCC and a dedicated runtime system for dataflow tasks. It is called OpenStream, and comes with a performance analysis tool called Aftermath, coupling high-level programming model concepts with low-level hardware counter statistics.

http://www.openstream.info

StarSs – StarSs is a task-based programming model that enables the exploitation of the applications' inherent parallelism at the task level. To mark the tasks in a StarSs application, annotations (pragmas) extending those of the OpenMP ones are used. A uniqueness of StarSs tasks are the input, output or inout clauses that applied to tasks' parameters enable the runtime to track tasks' data dependences. These pragmas have been adopted to form the dependent task construct of OpenMP 4.0.

http://pm.bsc.es/ompss

OpenHMPP - OpenHMPP (for Hybrid Multicore Parallel Programming) is a programming standard for Heterogeneous and many-core computing. Based on a set of compiler directives, it has been designed to handle hardware accelerators without the complexity associated with the low level programming. Compilers take C/C++ or FORTRAN code in input and generate the corresponding CUDA or OpenCL code and the integration with the original application.

http://en.wikipedia.org/wiki/OpenHMPP

OpenACC - Initially developed by PGI, Cray, NVIDIA, and CAPS entreprise, OpenACC is an open parallel programming standard designed to enable the usage of heterogeneous CPU/GPU computing systems. The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and FORTRAN to be offloaded from a host CPU to an attached accelerator. OpenACC is designed for portability across operating systems, host CPUs, and a wide range of accelerators, including APUs, GPUs, and many-core coprocessors.

http://www.openacc-standard.org

2 Executive Summary

We report on the finalization of the TERAFLUX compilation flow that took place in the fourth year, validating the integrated programming models, compilation algorithms and runtime systems targeting the TERAFLUX instruction set.

As a “Prototype” deliverable (P), we provide a survey of the achievements and match them to the goals of the associated task in WP4. We refer to associated documents, papers and online sources for a more detailed, technical discussion.

As planned in Task T4.4, we validated the complete flow from the StarSs efficiency language down to the TERAFLUX instruction set running on 1024 cores, considering realistic applications from WP2. We also conducted performance analysis and performance portability experiments.

The tool flow is distributed as free software, primarily as a patch to GCC 4.7.1, and with accompanying runtime libraries and tools. It comes with a complete set of benchmarks selected from the comprehensive list of applications characterized in WP2, and with tutorial examples introducing the language constructs of OpenStream.

Performance experiments on the TERAFLUX instruction set simulator, scaling to multiple node configurations, have been conducted on the different applications characterized in WP2.

- The tool flow for the efficiency programming model relies on the OpenStream language and its implementation in GCC 4.7.1. The backend of this compiler has been modified to generate T* instructions targeting the TSUF branch of COTSon. This backend compiler has been extended to support OWM. The systematic conversion of StarSs to OpenStream has been validated and applied to a number of TERAFLUX applications.

- The CAPS many-core compiler supporting the OpenHMPP and OpenACC standards has been adapted to support shared-memory manycore architectures, building on the experience of the TERAFLUX compilation flow and programming models. The development of the TF infrastructure implies the extension of existing programming models to support legacy application. This document describes the extension of the OpenHMPP programming language, and using this extension, the design and implementation of a prototype HMPP Workbench able to target TF resources. The study proposes a new HMPP Server platform able to provide a generic and powerful infrastructure for the development of multiple CPU targets using the HMPP Workbench compiler suite.

- The tool flow for the productivity programming model is based on Scala and a Scala to C++ compiler, and on the T*-extended backend of GCC. Detailed experiments with Scala transactional memory and dataflow libraries have been reported in previous years.

- A specific tool for the performance debugging of OpenStream applications has been designed and implemented, called Aftermath, which is distributed as part of the OpenStream platform. Performance portability experiments leverage GCC's intermediate representation for OpenStream programs and a polyhedral compilation flow adapted to the needs of dataflow architecture.

3 Introduction

The overall objective of WP4 is the development of compilation and runtime support tools tailored to the TERAFLUX architecture and programming models. The compiler(s) need to map the parallelism and locality as available from the source program and programming model to the target execution model and architecture. The distribution of the roles among the compilation tools and the runtime tools is guided by the efficiency and robustness of handling the challenges statically or dynamically, respectively.

The source program exhibits high levels of concurrency, but it still has to be exploited effectively on the target. The compiler tools need to coarsen the grain of synchronization, to issue bulk communications, overlap communication and computation, balance computation with communication bandwidth, and harness temporal locality of code and data, taking into account the features of the memory hierarchy. It also needs to generate tightly scheduled computation kernels, possibly targeting accelerators.

The deliverable also covers the design and implementation of a TERAFLUX specific flow for the OpenHMPP programming model proposed by CAPS entreprise. OpenHMPP offers a language, methods, and tools to express both the hybrid and the parallel execution of critical code sections on an accelerator. It is already used on by a large set of high performance applications and does not require a massive rewriting of the code for exploiting many-core architectures such as GPUs. The principles used by the OpenHMPP programming model can be applied with little modifications to the virtualization of computing resources and in particular to the programming of TF systems. A dedicated section recalls the basis of the OpenHMPP programming model available with the HMPP Workbench; it continues with the description of the impact of the prototype development in the design of the new compilation infrastructure able to address virtualized computing resources in complex systems.

We report on the achievements of the TERAFLUX project in Task T4.4. We succeeded in closing the gap between the efficiency and productivity programming models and a multi-node 1024-core execution using the TERAFLUX instruction set. We also provided tools for performance analysis and to enable the performance portability of dataflow applications.

3.1 Document structure

Section 4 surveys the main tools covered by this “Prototype” deliverable and refers to online sources with more detailed information.

Section 5 reports on the integration of the Scala tool flow with the backend compiler for execution on the TERAFLUX architecture.

Section 6 presents a tool for the direct translation of the C-code into T* extended code.

Section 7 is a more extensive descript on the HMPP and OpenACC tool flow for TERAFLUX.

Section 8 surveys the performance analysis and performance portability experiments conducted with OpenStream and polyhedral compilation methods.

3.2 Relation to other deliverables

This deliverable extends the compilation algorithms discussed in D4.4 and D4.6, and evaluates them on a range of representative benchmarks. It also complements D3.4 with a description of the Scala to T* compilation flow.

3.3 Activities referred by this deliverable

This deliverable is associated with and represents the results of Task 4.4.

4 Tools

The prototypes available at the end of the project year can be sorted in 3 categories.

- The first ones are Scala-specific and support the productivity programming model. They are based on runtime libraries from UNIMAN, described in D3.3 and D3.4, and on a C++ compilation flow for the execution of Scala programs on the TERAFLUX architecture.

  Technical information and code are available online and updated regularly: http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS

- The different efficiency programming models all come with their source-to-source compilation framework. The TFlux language and tools have been extended to support transactions (see D3.3). Locality optimizations and multi-level parallel programming extensions have been implemented in the HMPP Workbench 3.0 (now renamed "CAPS many-core compiler"). The dataflow extensions have been prototyped. A new framework supporting shared-memory manycore architectures has been designed and evaluated on the Intel® Xeon Phi™ coprocessor. In addition, in the same compiler, the OpenACC standard has been extended to support a high level dataflow extension described in WP3. The Mercurium compiler for StarSs is embedded into a comprehensive tool suite within the OmpSs infrastructure. Enhancements for multi-level parallelization and transactional memory have been integrated to the OmpSs tool flow, including the support libraries and code generators for different devices. Note that OmpSs and HMPP model the TERAFLUX architecture as an accelerator device.

  Technical information and code are available online and updated regularly: http://nanos.ac.upc.edu

- The TERAFLUX backend compiler is maintained as a patch to GCC version 4.7.1. It supports both native execution on x86 with a software runtime, and direct compilation of OpenMP dataflow streaming pragmas to T* extensions for execution on the TERAFLUX architecture. It supports Transactional Memory (TM) and Owner Writable Memory (OWM). It also supports parallel performance debugging/analysis, and performance portability experiments using polyhedral compilation.

  The final release of the prototype was made available to the partners and to the public parties on October 1st 2013. The distribution and maintenance are managed through an automated installation and source code repository process:

  http://www.openstream.info

  The development and exploitation of OpenStream will be continued in collaboration with UNIMAN, UPMC, and Kalray (the leading European low-power many-core processor company). Third party users include researchers at Ohio State University, and Politecnico di Torino. Technical information and code are available online and updated regularly.

5 Running Scala Programs on the TERAFLUX Architecture

The motivation for and the basic approach to translating from Scala to C++ were described in deliverable 4.6. To summarize, after two unpromising attempts to translate from Scala to executable code were abandoned, it was decided to translate from Scala to C++ using plugins to the Scala compiler. An existing experimental plugin produced by EPFL was identified which transformed the Scala AST into a form which was amenable to further processing into an executable form. A new plugin was then produced by UNIMAN which took the resultant AST and emitted C++ code.

At the time D4.6 was written, a pilot version of the translator had been produced which was able to handle small test programs. It was noted that the following features were not yet supported:

- Generics;
- Threads;
- Transactions;
- Exceptions.

When the provision of these features was investigated in the context of more complex programs, it became apparent that the approach used had limitations. A re-implementation of the translator plugin was therefore produced.

It became apparent that the use of Generics together with complex inheritance hierarchies made it very difficult to determine the types of variables to use in the translation. The pilot translator resorted to the use of `void*` in many circumstances, which resulted in total loss of type information. This in turn made dynamic binding very difficult or impossible in certain cases.

It was therefore decided to use the type inference mechanisms in C++11 as provided by the ‘auto’ feature. In addition C++ templates appeared to provide mechanisms for the implementation of generics as well as significantly reducing the amount of library code which needed to be produced. In the previous version of the translator, Scala basic types were mapped on to the corresponding C++ basic types. In the new version all basic types have been represented as objects to enable the full flexibility of the approach to be achieved.

5.1 Current Status

The new translator is able to deal with all the original features of Scala programs described in D4.6 together with those extras listed above.

The compiler can produce code for the following:

- Standalone Scala programs which can be translated to C++ and then directly compiled to standard executable code.

- Dataflow style Scala programs which can be translated to C++ and then linked with a software version of the TERAFLUX TSU (Thread Scheduling Unit). This uses pthreads to provide multithreading and, depending on the underlying machine and pthreads implementation, make use of multiple cores.

- Dataflow style Scala programs which can be translated as above but to target the TERAFLUX T* interface. This can then run on the COTSon simulator of the TERAFLUX machine. This provides full simulation of the support for Transactional Memory.

There are still a few limitations on Scala which can be handled. Much of this is simply a lack of implementation of library code which can readily be provided with further effort. The use of C++ templates, although successful in many respects, has added complexity in the implementation which was not anticipated and some of this may place restrictions on the range of Scala features which can be supported.

The translator has been evaluated on over 20 test programs exploring a wide range of features of Scala. Three applications have been used for an evaluation of the dataflow simulation route: i) a double recursive Fibonacci function to verify basic functionality, ii) a blocked Matrix Multiply to evaluate numerical performance and iii) a 3 dimensional (2 layer) Lee routing algorithm to exercise the use of transactions.

Unfortunately the new approach, although more flexible has proved less efficient both in terms of speed and memory usage. As a consequence, we have only been able to simulate relatively small versions of the benchmarks. This has led to limited parallel speedups to be achieved. For example a 64x64 blocked matrix multiply produces speedups of around 16 on 32 cores. In addition, the execution speed of the translated Scala, when run directly, has reduced by a factor of approximately 3 compared to the previous translator.

5.2 Ongoing work

It is believed that the majority of both the performance and memory issues are a consequence of the decision to use objects for basic types. It is believed that this can be reversed in the new translator with moderate effort and this is underway. It is hoped that a much more comprehensive evaluation can be performed and demonstrated before the TERAFLUX review meeting.

6 Direct translation of C-code into T* instructions

In the task T4.4 (last period), UNISI has continued to support the deployment of the T* ISA extension (cf. all WP6 and WP7 deliverables) in close cooperation with INRIA (cf. D4.3 for the initial T* mapping), and relying on the minimalistic T* instruction initially presented [12], i.e., mainly relying on the TSCHEDULE/TDESTROY, TREAD/TWRITE, TALLOC/TFREE instructions.

In relation to this task and general workpackage objectives, UNISI has explored the possibility to translate a generic C program (referring to the C89 specification) directly into programs that use the T* instructions.

This effort does not overlap with other efforts, but provides a practical demonstration of the theoretical ability of translating any program in a form that only exploits the initially devised DF-Threads (cf. D7.1). In particular, we focused on the translation of what we call "T*C" that we used in WP7 (cf. D7.4) in order to show how to use the COTSon simulator more easily rather than coding directly into T* assembly instructions.

The T* ISA allows us to: i) execute a subprogram and then another subprogram (sequence), dynamically, through the TSCHEDULE instruction; ii) executing two subprograms according to the value of a boolean expression (selection) through the predicate of TSCHEDULE; iii) executing a subprogram until a predicate is true (iteration), again through the predicate of TSCHEDULE. Therefore, by the Bohm-Iacopini theorem, any algorithm can be expressed using the T* instruction set extension so that the classical control structures are translated into dataflow dependences.

Please note that by releasing totally the need of synchronous calls, an asynchronous dataflow execution-model is then generated by relying on the T* ISA extension, while the programmer doesn't have to specify anything in the source C-code, thus allowing a large base of legacy code to be re-used. This effort also helped INRIA and BSC develop the algorithm used for the systematic conversion of StarSs programs to OpenStream: both problems boil down to the principles established in the Bohm-Iacopini theorem.

We are not making any claim on the quality of the generated code, in particular in respect to any parallelization that might be applied through tools like PLuTO or other polyhedral compilers. The methodology that we applied relies on the SCALE compiler, by modifying the backend of that research compiler in other to target first a pure dataflow representation of the program in term of DF-Threads, and then mapping such representation to the T*C. From the T*C representation (which is again C-code), we translate the code into x86+T* extension by relying on the already presented GCC toolchain (cf. D7.4). The generated programs can run directly on the COTSon simulator as modified thanks to this project, and on future machines that may support the T* extension.

The aim of this toolchain is therefore to provide an additional path for translating C-code into executable T* code, directly. This tool can be demonstrated on demand but it is not yet available for distribution.

7 HMPP and OpenACC Tool Flow for TERAFLUX

HMPP Workbench is a feature-rich tool that simplifies the utilization of GPUs and many-cores systems. This section provides the basic information about how to get started with the OpenHMPP directive set. We introduce:

1. The basic concepts of OpenHMPP (e.g. RPC, codelet, memory model, gridification);
2. The basic directives (e.g. codelet / callsite);
3. Code generation and optimization directives.

Furthermore, the HMPP Workbench supports the OpenACC [7] [8] standard directives – a parallel-programming standard announced at Supercomputing 2011 and supported by NVIDIA, Cray Inc., the Portland Group (PGI), and CAPS entreprise.

For more information, refer to the HMPP Workbench documentation [9].

7.1 Basic concepts

7.1.1 « Remote procedure call » (RPC)

HMPP allows to program hardware accelerators (HWA) using the Remote Procedure Call paradigm. Hardware accelerators commonly come in the form of discrete cards connected to the CPU through a fast bus interconnect like PCI-Express. This interconnect link is in the case of a Cloud target a network bus. In both cases, the bus shows two important properties: first, the accelerator uses a remote memory with a separate address space; secondly, the performance of the bus is significantly lower than the CPU memory system (up to several orders of magnitude). The RPC Protocol is used to proceed with the offloading of the computation from the host.

An RPC sequence, exposed Figure 1, consists of 5 steps:

1. Allocate the HWA and the memory;
2. Transfer the input data: Host => HWA;
3. Execute the sequence of the instructions;
4. Transfer the output data: HWA => Host; and
5. Release the HWA and the memory.

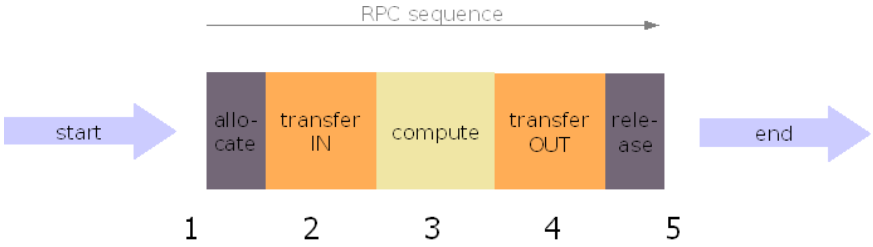

Figure 1, the five steps of an RPC sequence in HMPP

With HMPP directives, these steps may be performed somewhat implicitly or controlled using the relevant directives inserted in the source code.

7.1.1.1 Memory model

Host and devices usually do not reside in the same memory space (see Figure 2). The application and the HWA have their own private memory only accessible via explicit memory transfers. HMPP deals with this in a transparent way for the user; it can be seen as the glue between target-specific programming environments and general-purpose programming techniques.

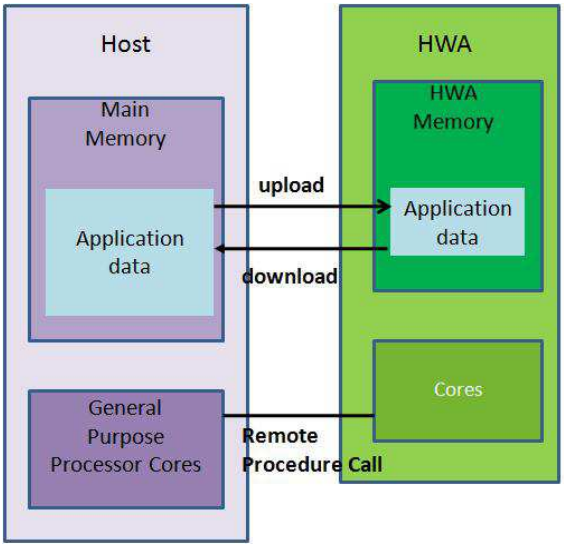

Figure 2, Memory model of OpenHMPP

The memory model can be completed depending on accelerators with some local memories that will have to be managed directly inside the codelet.

7.1.1.2 Codelet

A codelet is a computational part of a program located in a function inside the application. It takes several parameters, performs a computation on these data and returns. A codelet is typically a C function or a FORTRAN subroutine automatically translated by HMPP into HWA-specific code such as CUDA or OpenCL [10]. Parameters of the codelet defined the data allocated and manipulated on the accelerator. They can be specified as *IN*, *OUT*, *INOUT*, or *CONSTANT*. The execution of a codelet is considered atomic: the execution does not have an identified intermediate state or data.

A codelet has the following properties:

1. It is a pure function;
2. The number of arguments is fixed (i.e. no variable number of arguments like C vararg);
3. It is not recursive; and
4. Its parameters are assumed to be non-aliased.

These properties ensure that a codelet RPC can be remotely executed by a HWA. This RPC and its associated data transfers can be performed asynchronously. The codelet itself does not specify the parallelism but define the computation area deported on the accelerator.

7.1.1.3 Parallelism / Gridification

In the context of HMPP, "gridification" refers to the transformation of codelets' loop nests into computations that can run in parallel on a hardware accelerator such as a GPU. Gridification is about translating blocks of loop iterations into a set of parallel kernels. This translation is done by the conversion of the loop nest iteration space into a set of equivalent grid computations with as many dimensions as there are induction variables. The grid computation kernel is the minimal computational object. It is characterized by its number of dimensions (from one to three) and proceeds to the computation by the execution of multidimensional sub-blocks, with all the iteration space executed in parallel by the hardware. Sub-blocks can be executed out-of-order by one or multiple multi-processors until completion.

```
#pragma hmppcg gridify(i,j)
for( i = 0 ; i < n; i++ ) {
  for( j = 0 ; j < n; j++ ) {
    float prod = 0.0f;
    for( k = 0 ; k < n; k++ ) {
      prod += A[k*n+i] * B[j*n+k];
    }
    C[j*n+i] = alpha * prod + beta * C[j*n+i];
  }
}
```

Listing 1, Gridification of a matrix multiplication

The gridification process is controlled by a specific set of directives prefixed by "hmppcg". Combined with loop transformations, they control how the original loop nests are converted into grid computations. This optimization phase is critical to get the maximum performance from the accelerator architecture. In the example showed in Listing 1, the hmppcg gridify directive is used to set the <i, j> loop nest as parallel.

The gridification is complementary to other parallelization methods like the "vectorization" or the "multi-threading".

7.2 Basic directives

This section describes the two main directives needed to program the remote execution of a computation. More directives are described in the workbench manual.

HMPP directives follow the same principle as in OpenMP. Each directive starts, in C, with:

```
#pragma hmpp <commands and arguments ...>
```

In FORTRAN:

```
!$ hmpp <commands and arguments ...>
```

The pair of directives codelet/callsite is the minimum required to get a function to run on an HWA. The codelet directive must be placed on the function declaration, and the callsite directive must be placed on the occurrence

of the function call to be offloaded to the HWA. The codelet is declarative and lead to the generation of the code for the target accelerator (specified by the argument "target"). The function should contain parallel loops to get performance on manycore devices.

```
#pragma hmpp myCall codelet, args[*].transfer=atcall, target=CUDA
void myFunc(int n, int A[n], int B[n]) {
  int i;
  for (i=0; i<n ; ++i)
    B[i] = A[i] + 1;
  }

void main(void) {
  int X[10000], Y[10000], Z[10000];
  // ...
  #pragma hmpp myCall callsite
  myFunc(10000, X, Y);
  // ...
  myFunc(1000, Y, Z);
  // ...
}
```

Listing 2, Codelet/Callsite C example

The callsite directive is set before the function call of the declared codelet. This directive tells the HMPP compiler to change the original source code to operate the RPC of the codelet on the accelerator. The code in Listing 2 proposes a condensed view of the OpenHMPP program.

Real applications require the execution of multiple code sections on accelerators, so the HMPP programming model provides the definition of "groups" assembling a set of codelets and their arguments into one single object. Inside a group, codelets arguments are visible and shared. This mechanism reduces the memory consumption and the memory transfers on the accelerator.

7.3 Execution and compilation

A program with HMPP directives can be compiled by prefixing the original compiler command with the keyword "hmpp". Note that only sources codes with OpenHMPP directives, and eventually the final link, need to be completed this way. The regular building system can be left untouched for other parts of the application.

```
hmpp gcc basic_codelet_callsite_example.c -o tstC.exe
```

Listing 3, compiling a C source file with HMPP directives using the GNU C compiler and HMPP

An application compiled with HMPP – i.e. a codelet application – can be executed by simply running the resulting binary.

7.4 Consequences of experimentations on the TERAFLUX architecture

Experiments made on the implementation of a TERAFLUX target ("TF" for short in the following) in the HMPP Workbench have led to the design of a new software architecture for the HMPP compiler. The former approach, applied to OpenCL type languages, was relying as much as possible on tools and interfaces available

for these languages, and very often, also on proprietary API provided by accelerating boards: initialization and allocation API, parallel programming paradigms. The code generation and the runtime management in the CAPS compiler were significantly limited by these restrictions. They have been an important constraint in the implementation of a TF target in the compiler, but also in the implementation of various target in other fields (embedded processors, native support of the Intel Xeon Phi accelerator).

The new software architecture must offer the following features:

1. the decoupling of the parallel programming model from the hybrid programming,
    a. necessary to deport any type of parallel programming models,
    b. simplify the programming of the hybrid management;
2. the introduction of a boot protocol independent of the regular device allocation,
    a. takes in charge the upfront reservation of TF resources,
    b. takes in charge the authentication and secure access of resources;
3. an abstraction of communications,
    a. enables the implementation of serialization and un-serialization of complex data,
    b. enables the topology conversion of structured data (performance oriented),
    c. implements the secure data transfer configured by the boot protocol;
4. a dedicated and specialized runtime on the accelerator board,
    a. implements the bufferisation and the secure communication of asynchronous requests,
    b. can handle the allocation and the creation of resources according to the application workload;
5. the creation of an “execution context” independent for all codelet or parallel tasks,
    a. enables the implementation of interleaved RPC kernel calls (with the support of lightweight threads),
    b. enables the mixed implementation of different parallel execution models;
6. the creation of communication channels between execution contexts (in a hierarchical way),
    a. enables the workload distribution over a tree of computation units,
    b. enables the implementation of Active-Message communications.

All these features shall be implemented with in mind the best balance between two objectives: a possible generalization or extension to various types of architectures available on the market, and with a higher priority, the performance.

7.5 Definition of a new innovative deployment architecture for HMPP

The design of the new software architecture has led to the implementation of a new hybrid execution platform for general purpose CPU: “HMPP Server”. The objective of this module is to propose a unified code generation platform for high performance, hybrid, and massively parallel machines. It should be able to support all current and future hybrid or parallel architectures for both the OpenHMPP and the OpenACC programming models.

Major properties of the OpenHMPP programming models are kept: the code generated is preserved from the original application by the directive approach, the hybrid compilation chain is automatic and autonomous, and

finally, the model relies on the RPC execution of codelets, functions containing the code to be deployed with massively parallel sections.

7.5.1 HMPP Server Architecture

The architecture is presented Figure 3. It is composed of two major sections: the *HOST* section and the *DEVICE* section. The already existing modules in the architecture are in green, and the new modules are in orange. Hardware blocks are in yellow.

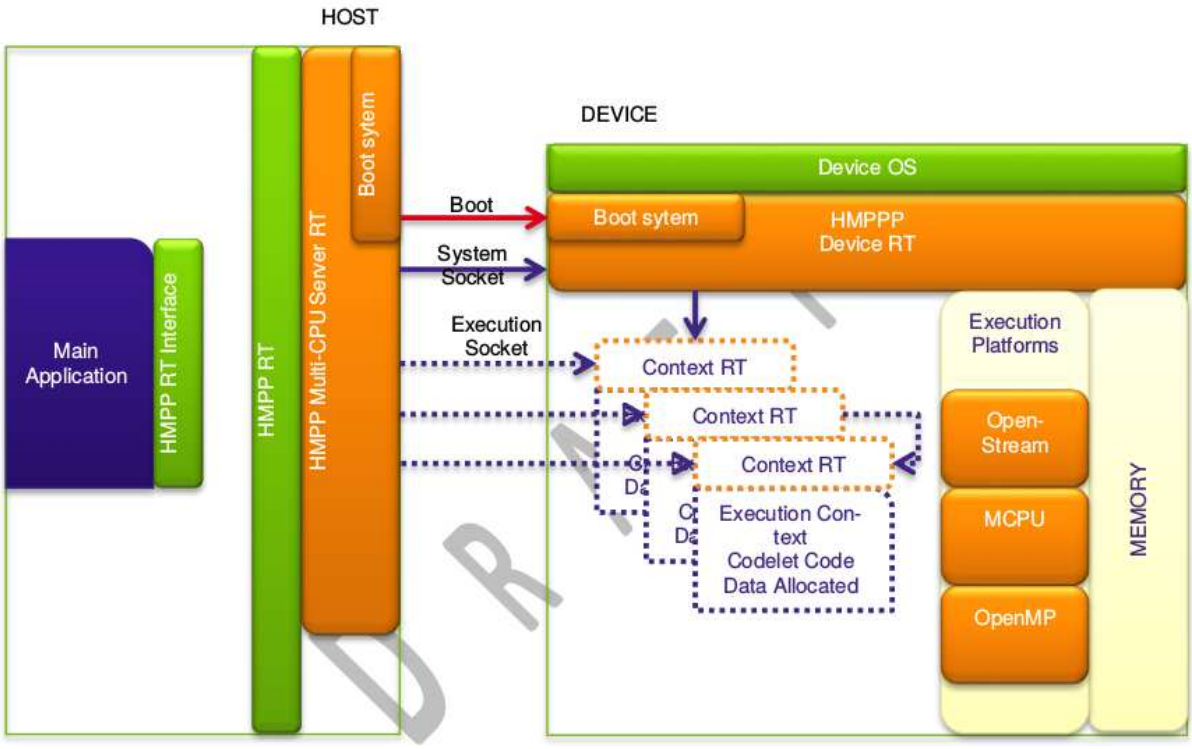

Figure 3, HMPP Server Architecture

7.5.1.1 Host section

The host section takes in a large part the original design of the HMPP Workbench, with an instrumentation of application's directives with runtime requests, and the usual HMPP runtime. We added the target specific runtime part, in our case the generic Multi-CPU runtime, and a new boot module for some new accelerating targets like the generic Multi-CPU target.

7.5.1.2 Accelerator management

The device section is totally renewed. First of all, the boot module is the only element that can be placed out of the accelerator runtime: it can be placed depending on cases on the host side, or in an external and remote machine in charge of the global administration of resources. Its duty is to answer all requests of creation or allocation of devices at a global scale.

Then, an accelerator runtime is implemented on the accelerator itself. Its implementation has to be conformed to the constraints and to the specificities of the machine: cooperation with the device OS, adaptation to available API. A target can enforce restrictions about the execution models available on the device. The boot module and the accelerator runtime are the basis of the virtualization of the device, and of the dynamic management of resources (data or computations).

Communications between runtimes and accelerator codes are done through "Socket" type interfaces: the channels. They can be quickly implemented using any interconnection protocol available on the system: TCP/IP or UDP/IP sockets, DMA channels, SSL tunnels, Infiniband interconnects, etc. In order to achieve the best performances, different channel types can be used depending on the type of connection and the type of target. System channels for the Boot module (socket boot) shall be robust and efficient for control type messages; runtime command channels (system channels) should have low latencies; operational channels (execution sockets) should have both low latencies and high bandwidth using preferably a specialized hardware (e.g., DMA). All channels have to support natively the serialization and optionally the secure transmission of data.

7.5.1.3 Execution management

The execution model is composed of a various parallel code generation back-ends, each proposing to the user a different parallel programming paradigm. They are all designed to extract the maximum performance for one specific machine architecture with a specific input code. Execution schemes can vary from a pure task parallel code execution, to a massively parallel execution in grids. A sequential back-end generating a standard "C" code is available for basic hybrid or remote executions and can be useful for validation. The OpenMP back-end can be used with parallel loops to exploit the parallelism using the native OpenMP target compiler: this one is of the simplest way to get performance with parallel code on multi-core targets. The OpenStream compiler can be used as well in the same way. The MCPU back-end is a massively parallel execution scheme in grid designed to extract performance on multi-core machines with lightweight threads.

When generated, the code is sent on the accelerator with its own execution context. This context is linked to the application with a unique operational execution channel. From this context, the code can create new regular or simplified contexts. In the case of a regular context, a complete operational execution channel will be created and connected with its parent. In the case of a simplified context, the link will be made with Active-Messages. By construction, all contexts are asynchronous and are executed in parallel. The global execution creates a hierarchical distribution of the computation with eventually different local execution models. This architecture permits the implementation of complex recursive and parallel computations with an automatic workload adaptation, and a better scalability for large systems.

7.5.2 Implementation on the Xeon Phi architecture using low level API

The new architecture has been implemented on a beta branch of the HMPP Workbench version 3.0.8 for the low level support of a Xeon Phi accelerating board.

At time of development of this prototype, the access to the OpenCL library and compilers for the board was not available: the OpenCL programing model usually simplifies the programming of this type of machines. Only low level API could be used to connect and generate both hybrid and parallel codes on Xeon Phi. The new

HMPP Server architecture has been an opportunity for the implementation this new target in the HMPP workbench, and this machine, enabled us the evaluation and the validation of the architecture on a real case.

Low level API were on one side the COI library used to access and initialize the Xeon Phi, and on the other side the proprietary SCIF library capable of performing fast transfers using hardware DMA. The native compilation chain supports OpenMP.

At the end of the implementation phase, the architecture could be nearly completely validated. The validation could qualify:

1. the management of the unified boot system;
2. the management of the compilation chain and of the hybrid code generation;
3. the preliminary support of execution contexts and of interleaved kernel executions;
4. the support of boot, system, and operational channels by a proprietary DMA.

The full chain of the beta version of the HMPP server was validated – from the directive programming to the execution on the device – using the “HydroC” application. This real life C application performs a simulation of hydrodynamic flows using the “Godunov” algorithm. The validation was performed on two versions of the application: a first version using OpenHMPP (version 2 and 3) and a second version using the OpenACC language. In both cases, the application used the OpenMP code generation back-end.

8 Performance Analysis and Portability Experiments

So far, the project's activities in WP4 have focused on optimizations and compilation flow to support dataflow programming and transactional memory. One long-term objective of advanced compilers remains to offer one more level of abstraction to programmers: performance portability. Loop tiling and thread-level parallelization are two critical optimizations to exploit multi-processor architectures with deep memory hierarchies. These optimizations are some of the most effective ones, when exposing coarse grain parallelism, to adapt the program to a given architecture and execution context.

The dataflow model of execution does not involve the scalability drawbacks of barrier-based parallelization: tasks can execute as soon as the data becomes available (i.e., when dependences are satisfied) and lightweight scheduling heuristics exist to improve the locality of this data in higher levels of the memory hierarchy. But adaptation and advanced optimizations remain key to reach efficient levels of performance.

Loop transformations for the automatic extraction of data parallelism have flourished. Unfortunately, the landscape is much less explored in the area of task parallelism extraction and in particular the mapping of tiled iteration domains to dependent tasks. We describe below the three key contributions:

- Experimental: we demonstrate strong performance benefits of task-level automatic parallelization or re-optimization of parallel programs over state of the art data-parallelizing compilers, and we further characterize these benefits within and across tiled loop nests.
- Algorithmic: we design a task parallelization scheme following a simple but effective heuristic to select the most profitable synchronization idiom to use. This scheme exposes concurrency and favors temporal reuse across distinct loop nests (a.k.a. dynamic fusion), and further partitions the iteration domain according to the input/output signatures of dependences. Thanks to this compile-time classification, much of the run-time effort to identify dependent tasks is eliminated, allowing for a very lightweight and scalable task-parallel runtime.
- Compiler construction: we implement the above algorithm in a state-of-the-art framework for affine scheduling and polyhedral code generation, targeting the OpenStream research language.

We use the Ring-Roberts kernel below as an illustrating example, with N=4000 and using double precision floating point arithmetic.

```
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++)
S1: B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j] +
    A[1+i][j] + A[i-1][j] + A[i-1][j-1] +
A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;

for (i = 1; i < N-2; i++)
  for (j = 2; j < N-1; j++)
S2: A[i][j] = abs(B[i][j]-B[i+1][j-1]) + abs(B[i+1][j] - B[i][j-1]);
```

Table 1 below shows the performance obtained when using ICC -O3 -parallel as the compiler. The original code reaches 1.2 GigaFlop/s (GF/s) on an AMD Opteron 6274 (16 cores, 16x16KB L1, 2 GHz) and 2.7 GF/s on an Intel i7-2600 (4 cores, 4x32KG L1, 3.4 GHz). This program is memory bound but contains significant data reuse potential, so tiling is useful to improve performance. We leverage the power of the PLuTo compiler to tile simultaneously for coarse grained parallelism and locality, considering the two fusion heuristics "min-fusion" (do not fuse loops) and "smart-fusion" (fuse loops with common dimensions).

Table 1: Performance of Ring Roberts on Opteron and Core i7

Proc-cores	ref ICC	pluto minfuse	pluto smartfuse	our work
opt-1	1.25	0.4	0.7	0.9
opt-8	1.25	2.7	3.9	4.7
opt-16	1.25	2.0	0.7	6.8
i7-1	3.4	2.6	2.3	2.8
i7-2	4.2	3.6	4.0	5.4
i7-4	4.1	3.5	4.3	10.1

One strategy, referred to as “minfuse” in the PLuTo compiler, tiles and parallelizes each loop nest independently, requiring 2 barriers. The symmetric strategy, referred to as “maxfuse”, attempts to fuse loops even at the expense of resorting to more complex sequences of loop transformations such as loop skewing, pipelining and peeling to enable the fusion of program statements under a common loop nest. The “maxfuse” strategy exhibits an outermost sequential loop and a second outermost parallel loop followed by its implicit barrier. However, the performance does not increase linearly with the number of processors, instead the performance either reaches a plateau with the Intel i7, or drastically drops in the Opteron’s case.

Figure 4 illustrates the nature of data dependence in the Ring-Roberts code and the parallelization options for static affine scheduling, as represented by state-of-the-art polyhedral compilers like Pluto, versus dynamic task dataflow. The top half of the figure illustrates the iteration spaces for an unfused form of the code, with the left square representing the first loop nest and the right square the second loop nest, along with 1D tiling of each loop nest, working on blocks of rows of the matrices. With Pluto minfuse, a barrier is used between execution of tiles of the first and second loops. With smartfuse, the two loop nests are fused, but skewing is required to make fusion legal. But after fusion, only wavefront parallelism is feasible with 2D tiling (and no parallelism with 1 D tiling), with barriers between diagonal wavefronts in the tiled iteration space. Thus there is a trade-off: with min-fuse, the tiled execution of each loop nest is load-balanced, but interloop data reuse is not feasible; with smart-fuse, inter-loop data reuse is exploited, but load imbalance at start-up results for tiled wavefront parallelism.

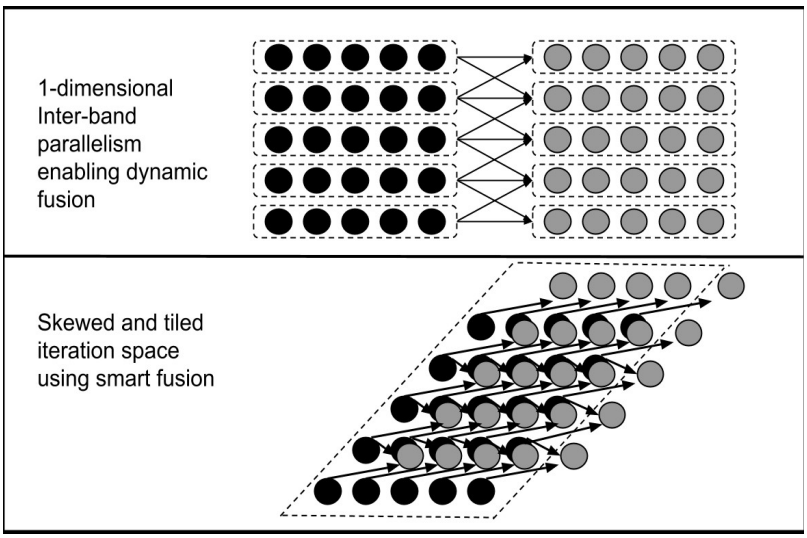

Figure 4: Task-parallel data flow over static affine scheduling

With a task dataflow model, it is possible to get the best of both: increased degree of task-level parallelism as well as inter-loop data reuse. This occurs by creating 1D parallel tasks for each loop, but the point-to-point task level synchronization enables ready tasks from the second loop nest to be executed as soon as the needed tasks (the ones corresponding to the same block row and the ones on either side) have completed. Thus, a "dynamic fusion" between the loop nests is achieved automatically, without the problem of load imbalance from the wavefront parallelism with a static affine tile schedule for the fused loop nests.

This problem has been recognized in the linear algebra community and specialized solutions have been designed. We propose a general-purpose solution by leveraging properties of the schedule of tiles as computed by polyhedral compilers, and utilize it to determine at compile-time the inter-tile-band and intra-tile-band dependences. Unlike classical approaches in automatic parallelization, these dependences will then be instantiated at run time, and fed to a dynamic task scheduler. The last columns in Table 1 show the performance obtained when generating task-parallel code from the PLuTo min-fuse heuristic when compiling the task body with ICC -O3 xAVX. As one can see, after optimizing the task body, we obtain near perfect scaling on Intel's i7, yielding a 4x and 2x improvement, over ICC and PLuTo's best, respectively; whereas on AMD's Opteron we obtain over 6x relative to the baseline and 2x over PLuTo.

Our technique can be summarized as follows. We first compute tile-level constructs which are the input to an algorithm that selects stream idioms to be used for each tile dependence or to a partition routine which splits the loop nests into classes that share identical input/output dependence patterns. This algorithm chooses when to extract parallelism across disjoint loops, while the partition routine allows to create a dynamic wavefront of tiles. Then a (static) task-graph is constructed to prune redundant dependences and to decorate it with dependence information. Finally, code is generated for the OpenStream run-time.

We conducted systematic performance evaluations on the Polybench suite, comparing task-parallel dataflow execution with barrier-based data-parallel versions, on 3 different multicore architectures. The results consistently establish the scalability advantages and performance portability of the task-parallel version.

9 Conclusion

We surveyed the final, integrated flow of the project, combining compilation methods and tools, and runtime systems, mapping both plain-C and modern efficiency languages such as StarSs, HMPP, OpenACC and OpenStream to the TERAFLUX execution model and instruction set. The flow also supports the conversion of a high-level productivity language, Scala, for efficient execution on a dataflow architecture. The development of a prototype of the HMPP workbench for the TF machine has led to the design of a new compilation infrastructure capable of addressing a wide range of accelerating devices using the OpenHMPP programming models. Major work has been necessary for the design and the implementation of new runtime systems to support the remote execution of codes on new systems like TERAFLUX machines. The new HMPP Server compilation framework offers a generic platform for the deployment of hybrid and parallel applications, and proposes a much more powerful infrastructure than the previous HMPP Workbench: it provides an extended support of complex resources, the securisation and authentication of data transfers, and extends the management capabilities of parallel computations.

Strong publication output and open source tool distribution was achieved as a result of the activities of WP4. Performance analysis and portability studies enabled by this tool flow have been demonstrated, with direct experimental validation on 1024 multi-node TERAFLUX simulation, the results being reported in WP2 and WP7.

10 References

[1] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and dataflow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO), selected for presentation at the HiPEAC 2013 Conference*, January 2013.

[2] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Shenzhen, China, February 2013.

[3] "Moving Scala ASTs one step closer to C, by turning them into three-address form", Scala Compiler Corner, http://lampwww.epfl.ch/~magarcia/ScalaCompilerCornerReloaded

[4] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG, associated with HiPEAC), Vienna, Austria, January 2014.

[5] Andi Drebes, Karine Heydemann, Antoniu Pop, Albert Cohen, and Nathalie Drach-Temam. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. ACM Transactions on Architecture and Code Optimization (TACO), 2014. To appear.

[6] Martin Kong, Albert Cohen, R. Govindarajan, Antoniu Pop, Louis-Noël Pouchet, P. Sadayappan. Compiler/Run-Time Framework for Dynamic Data-Flow Parallelization of Tiled Programs. Submitted for publication.

[7] NVidia, "NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing," 11 14 2011. [Online]. Available: http://pressroom.nvidia.com/easyir/customrel.do? easyirid=A0D622CE9F579F09 &version=live&prid=821214 &releasejsp=release_157 [Accessed 01 09 2012].

[8] HPCwire, "CAPS Entreprise Now Supports OpenACC Standard," 02 05 2012. [Online]. Available: http://www.hpcwire.com/hpcwire/2012-05-02/caps_entreprise_now_supports_openacc_standard.html.

[9] CAPS entreprise, HMPP Directives Reference Manual, Version 3.2.0, 2012.

[10] The OpenCL Specification v1.1 r36, "The OpenCL Specification," 30 9 2010. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[11] R. D. S. B. Francois Bodin, "Hmpp: A hybrid multi-core parallel programming environment," GPGPU Workshop, 2007.

[12] Roberto Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", ACM Computing Frontiers, Cagliari, Italy, May 2012, pp. 303-304, doi 10.1145/2212908.2212959