An OpenMP backend for the Σc streaming language

S. Louise

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 1073–1082

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

An OpenMP backend for the ΣC streaming language

Stéphane Louise[1]

CEA, LIST,
Nano-Innov, PC174, Bat. 862, 91191 Gif-sur-Yvette Cedex, France
stephane.louise@cea.fr

Abstract

The ΣC (pronounced "*Sigma*-C") language is a general purpose data-flow language that was initially targeted for Kalray's MPPA embedded many-core processor. It is designed as an extension of C, allowing the Cyclo-Static Data-Flow (CSDF) model of computation. Until now, it was only available for the first generation of the MPPA chip. In this paper, we show how we built an OpenMP back-end for the ΣC language, and we used this compiler to evaluate some of the assets of stream programming and some limitations of the current implementation, by evaluating the performance on several benchmark programs. This new back-end could open the way to utilize this language to study embedded stream programming concepts or to program HPC applications

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Cyclo-Static Data-Flow, streaming language, compilation, runtime generation, OpenMP

1 Introduction

In embedded computing, parallel computing is becoming the standard, for the sake of both performance and power efficiency. Nonetheless, usual HPC programming models are not well fitted, because they are memory and power hungry. High Performance Embedded System must rely less on memory coherence and more on explicit data-movements, with an easy way to avoid both parallelism and performance pitfalls. It should also be amenable to real-time constraints.

Hence, the most obvious way is to find a programming model that would fit these constraints. It should be hierarchical for easy design and bug tracking, deterministic, would work with any communication means including NoCs and shared memory. This gave root to a renewed interest for stream programming and data-flow concepts which started at the end of the 1990s but flourished in the mid 2000 decade. It is also the way we followed within the project in collaboration with Kalray, when it was time to think about a well fitted programming language for an embedded many-core[1].

[1]Kalray's MPPA-256[6] was the first many-core embedded processor designed in Europe with more than 256 cores.

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.251

1.1 Data-Flow and Stream programming

Dataflow concepts are not new. One of the base model is Kahn Process Networks (KPN), defined by G. Kahn in 1974 [11]. KPN provides a very simple model of computation which can be represented by a directed graph (P, E, T) where $P = \{p_0, p_1, \cdots, p_m\}$ is a set of processes *i.e.* something which can do computations, $E \subset P \times P$ is a set of directed communication edges between processes, and T the set of tokens that models data exchanges. The communication edges are First-In First-Out (FIFO) pipes, in which data are modeled as token production and consumption. Any process can consume a number $n \in \mathbb{N}$ of data tokens on any of its input channels and produce any number $n' \in \mathbb{N}$ of tokens on any of its output channels. The only constraint is that reading on an input channel is blocking if not enough data tokens are present.

It is simple, it is local (the data path are explicit, so all the required data for computations must be fed to the process through FIFO, it forbids global variables, and does not require memory coherence mechanisms). It is hierarchical as processes themselves can be decomposed into subgraphs (*i.e.* a part of a data-flow graph). It is locally deterministic while being parallel by essence. Yet, it lacks some desirable properties.

For embedded systems, the highest problems which remains with KPN is the lack of insurance against deadlocks (*e.g.* if a given process tries to read tokens on a channel which is never fed), and against memory overflows (*e.g.* if a given process keeps writing on a channel which is never read or insufficiently read), and none of this is decidable before runtime. This is why other models of computations, based on KPN, but with more restrictions on the input and output rates were designed to cope specifically with these issues.

The two most famous models are called Synchronous DataFlows (SDF) [12], and Cyclo-Static DataFlows (CSDF) [9]. SDF is very simple: the number of data tokens produced and consumed on input and output channels of any process are fixed for the whole life of the application. CSDF is a superset of SDF: The number of read and written tokens can vary in a cyclic way. Each process is subdivided in subtasks with fixed productions and consumption, and the firing of subtasks within a process obey a predetermined cycle. This is illustrated in Figure 1.

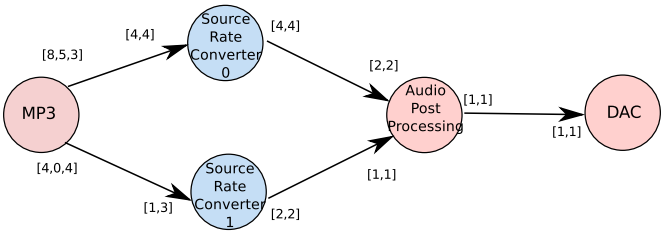

Figure 1: A simple example of a CSDF graph on a MP3 decoding application. The CSDF aspect can be seen e.g. on the MP3 process which is divided in 3 subtasks with different productions.

1.1.1 Extended features of the model

Languages like Brook [4] and StreamIt [2] brought 2 interesting features to SDF and CSDF formalisms:

- The ability to read on a given input channel a fixed maximum number of data tokens without consuming them. This allows for a simple implementation of sliding windows on signals and images which is a classic way to do signal and image processing,

- Some specific predefined generic processes do not perform transformations on data but only reorder the stream of data. The usually defined transformations include:

Splitters are CSDF processes with 1 input and $n \in \mathbb{N}$ outputs, and one integer $t_i \in \mathbb{N}$ per output. Each time a data token of order l is received on the input channel, then the data token is outputted on the output channel k such that $\sum_{i=0}^{k} t_i \leq l \mod (\sum_{i=0}^{n-1} t_i) < \sum_{i=0}^{k+1} t_i$. The usual case is when $t_i = 1, \forall i \in \{0, \cdots n-1\}$, so the first received token is outputted on channel 0, the second on channel 1, *etc.* This is a good way to enter a parallel section using a Single Program Multiple Data (SPMD) type of parallelism.

Joiners are the symmetric of Splitters, with n inputs and 1 output. The process waits for t_i token on its input line i and outputs the tokens on its output channel, then switches to wait for t_{i+1} tokens on channel $i+1$, until it comes to channel $n-1$ after which it returns to channel 0. This is useful to close a set of parallel treatments, either SPMD initiated by a splitter or a SDMP initiated by a duplicater.

Duplicaters are simple SDF processes for copy. They have 1 input line and n output lines. Each time a data token is read on the input line, it is copied on every one of the n output channels. It is useful to implement Multiple Programs Single Data (MPSD) type of parallelism.

As we found out with Pablo Oliveira [7], data distribution filters can be utilized to create any repetitive series of data of a given data-set, and if the compiler is aware of these filters, it can optimize them to the specificities of both the application and the underlying hardware system on which the application will run, even if the source implementation of the data distribution mesh was written in an inefficient way.

1.1.2 Repetition vectors and graph consistency

Looking at the graph illustrated on Figure 1, only very careful changes to the rhythms of productions and consumptions of data tokens preserve a consistent graph, *i.e.* a graph for which it is possible to find a static schedule that ends the graph (especially the communication FIFO) in the same state it started. When such a schedule exists, there exists a so-called *repetition vector* whose components are the number of time each task should be fired to come back to the initial state. For consistent CSDF graphs, there are 2 repetition vectors, one at the task level, and one at the subtask level. The task-level vector is in the kernel of the topology matrix of the graph, pondered by the consumption/production ratio. For subtasks, each component of the task-level repetition vector must be multiplied by the number of subtasks comprised in the associated task. See [9] for more details. For the graph of Figure 1, a task-level repetition vector is $\begin{pmatrix}1 & 2 & 2 & 4 & 4\end{pmatrix}$ and the subtask-level one is $\begin{pmatrix}3 & 4 & 4 & 8 & 8\end{pmatrix}$.

1.2 Programming manycores: The ΣC language

An obvious limitation of the StreamIt language that was at this time the leading research framework for embedded data-flow languages, is the type of graph a programmer is allowed to create with the language. The data-flow structure of StreamIt is mostly embedded in its syntax. It is created by declarative pre-determined structures of data distribution processes: A user process (or a *filter*, according to StreamIt semantic) always have at most one input channel and one output channel. Only predefined data-distribution filters overcome this limitation. These are mostly the Splitter, the Duplicater, and the Joiner. Moreover, Splitters and Joiners are always paired, as Duplicaters with Joiners, so that the combination is always equivalent to a filter with at most one input and one output. This leads to series-parallel graphs that are quite

```
subgraph root(int width, int height) {
  interface { spec {}; }
  map {
    agent output = new StreamWriter<int>(ADDROUT, width *
        height);
    agent sy1 = new Split<int>(width , 1);
    agent sy2 = new Split<int>(width , 1);
    agent jf = new Join<int>(width , 1);
    connect (jf.output, output.input);
    for (int i=0; i < width ; i ++) {
      agent cf = new ColumnFilter (height);
      connect (sy1.output[i], cf.in1);
      connect (sy2.output[i], cf.in2);
      connect (cf.out1, jf.input[i]); }
  }
}
```

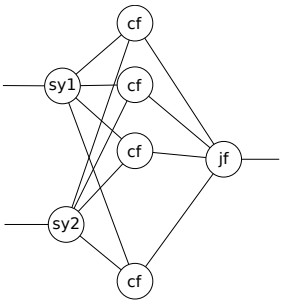

Figure 2: Topology building code, and the associated ΣC subgraph, showing multiple column filters (cf) connected to two splits (sy1 and sy2) and one join (jf). This type of construct is not possible in StreamIt.

limited in expressiveness. For this reason, the language also has to provide a special filter called *Feedback loop* to authorize the use of loops.

To overcome these limitations, we defined a two step compilation process in ΣC. The first step of compilation takes the definition of so-called *subgraphs* i.e. a part of the communication network, and the definitions of the filters (which could have any arbitrary fixed number of input and output channels) into a first program that is executed only to build the CSDF graph of the application. A mandatory first *subgraph* called *root* takes the same role as the *main()* function in C: It is an entry point to build the graph. It has no inputs and no outputs because I/O are managed outside of the CSDF formalism. A example of a subgraph as can be seen in Figure 2.

The Turing completeness of the C language allows to build any topology of CSDF graphs, but at the inconvenience of being a bit less easy to read for **simple** graphs compared to StreamIt constructs. Nonetheless, it is relatively easy to circumvent this limitation either by the use of the pre-processor or by using higher level Domain Specific Languages (DSL) to generate the topology. For Splitters, Duplicaters and Joiners, predefined agents called *system agents* (compiler-aware agents) can be used and their utilization is encouraged, because they can be efficiently optimized by the compilation tools. They are called respectively *Split, Join, Dup.*

The remaining parts of the code of the agents is otherwise copied mostly verbatim in the target C-code with the exception of FIFO accesses which are adapted to the target and the compilation choices. As a last aspect, the *interface* part of agents and subgraphs specifies the CSDF behavior of the so-called subgraph or agent. Figure 3 shows the details of a simple agent to operate filtering on a 2D picture.

As can be seen, the entry point for code execution of the agent is the *start* function. The associated *exchange* part indicates the associated input and output FIFO reading and writing which can be different than the CSDF consumption and production on the channel, thus allowing for sliding windows of data.

1.3 An OpenMP backend for ΣC

The idea of an OpenMP backend for ΣC seems at first a paradox. Nonetheless, as the language is not supported anymore by Kalray, we needed a way to keep it alive in our laboratory.

```
agent ColumnFilter(int height) {
  interface {
    in<int> in1, in2;
    out<int> out1;
    spec { in1[height]; in2[height]; out1[height]}; }
  void start () exchange (in1 a[height], in2 b[height], out1 c[
      height]) {
    static const int
    g1 [11] = {-1, -6, -17, -17, 18, 46, 18, -17, -17, -6, -1} ,
    g2 [11] = {0, 1, 5, 17, 36, 46, 36, 17, 5, 1, 0};
    int i, j;
    for (i=0; i < height; i++) {
      c[i] = 0;
      if (i < height - 11) for (j=0; j < 11; j++) {
        c[i] += g2[j] * a[i + j];
        c[i] += g1[j] * b[i + j]; }
    }
  }
}
```

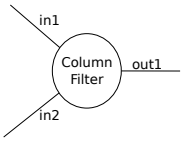

Figure 3: The ColumnFilter agent used in Figure 2 with two inputs and one output, and the associated portion of ΣC graph. It consumes 1 column of *int* on channel *in1* and on channel *in2*, and produces 1 column of *int* on channel *out1*.

With the powerful optimization steps incorporated to the compiler, and with an underlying model of computation that is very parallel-friendly, we should be able to produce OpenMP code that takes advantage of data-parallelism as well as pipeline-parallelism, opening the way to good performance, if it is not slowed down by the OpenMP runtime or the memory coherence mechanisms.

The remaining of this paper are the following: section 2 explains the source to source compiler architecture, section 3 show the strategy used to generate OpenMP code in the backend, section 4.2 show some preliminary results of our backend, and section 5 discuss related works before concluding.

2 ΣC compiler architecture

The ΣC compiler architecture is organized in 4 layers called by the scc (sigma-c compiler) script:

1. The front-end compiler that takes any input text file to check its grammar (ΣC is a superset of the C language) and generates 3 kind of files: the skeleton is the processing part of any agent (subgraphs do not have skeletons), the data file gather all the data that are generated by the graph generation, and the graph-generation program.

2. The graph instancing step gathers all the graph-generation programs, compiles them and runs them to generate 1) the CSDF graph and 2) the value of initialized data within the graph. It includes a first optimization step that removes irrelevant *system agents* like *split*, *join* or *duplicate*, so that these processes that do not alter the values are replaced whenever possible with simple pointer arithmetics [5].

3. The graph optimizer, partitioner, and pipeline scheduler. This step is already architecture specific, as it creates optimized partitions of the instanciated graph by taking into account the available memory, the communication ratio between tasks, and to create a mapping of the graph on the architecture and the routing of communications. It also calculates the repetition vector for each partition.

4. The runtime generator, and linker. It creates the specific runtime elements (buffers for FIFO, communication tasks, etc.). In the case of Kalray's MPPA, it also creates time vectors for dynamic scheduling, then it creates the binary, linked with a specific bare-metal execution support.

For more details about the implementation of the ΣC compiler for Kalray's MPPA, one can refer to [3].

ΣC for MPPA has been benchmarked against the other languages available for Kalray's platform and found to be close to the best in performance and surprisingly better for energy efficiency, on par with specific FPGA developments [10]. Nonetheless, as the ΣC language for MPPA did not had enough development drag, it is unsupported for the newer Kalray chips. This is one of the main reason to develop an OpenMP backend for the ΣC compiler.

3 Implementing an OpenMP backend for ΣC

There are several possible choices to implement an OpenMP backend for the ΣC language. We chose to avoid touching the first steps of compilation (particularly the first and second steps) as they are platform independent. Then we removed the specific optimization for the MPPA chip as part of the third step, so that it only generates the repetition vector.

Then comes the step of runtime and execution code generation which is platform specific, and was rewritten from scratch. We identified two possible choices. A first possible one is to remain close to the CSDF model and use the task model defined in version 3 and later of OpenMP. Such a choice has two drawbacks –a complexity of the generated code and uncertainties about the level of support of this feature in C compilers– and one possible advantage –a possible efficiency of execution for task-optimized OpenMP runtimes. The second possible choice is to remain close to the original thread-level parallelism of OpenMP, which would lead to a much simpler generated code, easier to generate and validate, and an average performance probably more or less independent from the utilized C-compiler. This is this latter approach we chose.

3.1 OpenMP execution code

The underlying principle is simple: using the repetition vector, we generate parallel sections that call the start function of each process the number of times it appears in the repetition vector. Using graph of Figure 1, its task-level repetition vector is $\begin{pmatrix}1 & 2 & 2 & 4 & 4\end{pmatrix}$, so the associated execution code would be as follows:

```
int main() { while(1) {
  for(int ipipe= 0 ; ipipe < 2 ; ipipe++)
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {_SigmaC_MP3_task1_start(ipipe);}
      #pragma omp section
      {for(int ip2= 0 ; ip2 < 2 ; ip2++) _SigmaC_SRC0_task2_start(ip2+ipipe*2);}
      #pragma omp section
      {for(int ip3= 0 ; ip3 < 2 ; ip3++) _SigmaC_SRC1_task3_start(ip3+ipipe*2);}
      #pragma omp section
      {for(int ip4= 0 ; ip4 < 4 ; ip4++) _SigmaC_APP_task4_start(ip4+ipipe*4);}
      #pragma omp section
      {for(int ip5= 0 ; ip5 < 4 ; ip5++) _SigmaC_DAC_task5_start(ip5+ipipe*4);}
    }
    #pragma omp barrier
  }}
}
```

The parameter of the **_start* functions is tied to the advancement index in the communication buffers that mimic the FIFO channels. In fact, a part of the scheduling is hidden in the indexation of the communication buffers. The *ipipe* index is a pipelining index so that while one given agent write into a part of the communication buffer, the agent on the other side of the communication channel reads in the other part of the buffer. The barrier at the end ensures that the synchronization of the execution is done once the repetition vector for the task is scheduled. The barrier is a point of synchronization, and therefore a place where the execution can loose performance, as the most obvious drawback of this implementation.

This effect can be mitigated by two factors: first, instead of the minimal repetition vector, one can choose any multiple of the repetition vector without other issue than increasing the size of the communication buffers. This is a way to play on the pipeline parallelism. The second one can be used for stateless agents, then the parallelism can be simply increased by using *omp parallel for* for each execution loop of these agents. It is worth noting that is what we did for the evaluation in the following section whenever we had stateless agents.

3.2 Agent instantiations

As seen in the listing, an instantiation of the agent is done by indexing the start functions with the name of the source agent and the number of its instantiation in the graph provided by the second step of the compiler. The same is done for all the variables created in the first step of the compilation of a ΣC program, including the communication buffers which are indexed by the name and the index of all the agents that read or write in the buffer.

Sizing the buffer is a simple consequence of the repetition vector and the pipelining level.

3.3 Indexing buffers

The most subtle point of this code generation is to create a correct indexing of the buffers, so that the task that writes in the buffer do not overwrites data elements that other tasks are currently reading. This process is done in two steps:

1. Calculate the scheduling rank of all the tasks of the graph. This is done by the following algorithm: initialize the rank as 0, and the set of current ranked task as void. Second, remove all the inputs of agents which fulfill their requirements in number of data tokens on their inputs. Third, put all the source agents (i.e. those without input channels) in the set of current ranked tasks. Fourth, count the number of produced data tokens on all the output channels when executing the tasks of the set according to their repetition component. Fifth, increment the rank index, and go back to the second step until the set of remaining task is void.

2. Once the rank of all the tasks is known, indexing the buffer is done in the reverse order of the rank index, modulo the pipelining depth, so that any instance of an agent reads its input buffers where the writing task at the previous level of pipelining should be doing at the previous execution step. If the writer is not in the previous set, the index must be shifted accordingly.

This way of proceeding allows the C compiler to do an important part of the remaining optimization work.

3.4 Missing features and limitations

One of the known missing features that would be the most detrimental is an optimization step to aggregate very fined grain agents so that the overhead for launching threads is not the limiting factor for performance. This is specially true when the useful code of the agents is only a handful of assembly instructions like what we can encounter in bitonic sorts (conditional swapping), or matrix multiplications (multiply and accumulate).

4 A first benchmarking of the OpenMP backend of ΣC

For the first test of performance we only tested the OpenMP compiled code on usual workstation. Even if they are far cries from many-core processor (the test platform is a Xeon-Nehalem 4 cores and 8 MT), it should reveal some of the potential and the weak points of the OpenMP backend of the ΣC compiler.

4.1 Benchmark applications

We tested a total of 7 applications. Two of them are native ΣC applications that we use internally for testing: a Laplacian (256×256) filter and a Fractal generator. The other 5 applications are automatic transformations of streamIt applications into ΣC (as streamIt features a SDF model, the transformation can be quite easily be automatically done from StreamIt to ΣC, see [8] for details). The tested 5 applications from the StreamIt benchmark suite. The characteristic of most of the StreamIt applications is that the granularity of their parallelism is fine or very fine. A strong asset of the StreamIt compiler is the task aggregation step, which is lacking for the current version of the ΣC compiler. There, we can expect a weak performance of the ΣC compiler for these fine grained applications.

4.2 Results

We compared the results of the ΣC compiler when finalizing the compilation either with no OpenMP flags and with the OpenMP flags. This simple test shows a first hints of achievable performance with ΣC, how well it works on ordinary workstations, and where the compilation tools will require improvements. The results are summarized in Table 1.

Table 1: Results of sequential vs parallel execution of the generated code. The two first applications are native ΣC applications whereas the 4 other ones are StreamIt applications automatically ported to ΣC

Application	Total task number	Task number	Naive Parallelism ratio τ	Real τ	Sequential execution	OpenMP execution	Speed-up
Laplacian	16	13	1.44	114	26.7±0.3	7.15±0.10	3.7±0.1
Fractal	50	16	8	150	53.4±0.4	7.62±0.81	7.1±0.8
Matmult	96	39	13	216	0.605±0.043	0.39±0.09	1.55 ± 0.05
Audiobeam	38	18	4.5	8.4	0.450±0.071	0.49±0.04	0.92±0.05
FM Radio	100	45	5.6	5.6	1.35 ±0.020	1.70±0.40	0.61±0.15
FFT2	19	14	1.75	80	1.039±0.014	1.42 ± 0.6	0.74 ± 0.04

As can be seen, and as expected, the fine grained StreamIt applications do not have a good parallel execution, because the overhead of parallelism management is larger than the gain

of parallel execution (only pipelining make it work a bit). Native ΣC applications which are developed with this limit in mind are on the contrary well fitted to the parallel execution with OpenMP. The Laplacian filter displays a speed up close to the number of cores which is good, especially for an application whose naive parallelism is less than 2. The speed-up for the Fractal application is close to 8, hence it can take advantage of the Hyperthreading feature of the Xeon processor, which is a good achievement and because the program is embarrassingly parallel.

A quick test was made to compare absolute results of mostly classic OpenMP development with the results from the Sigma-C compiler. The results on the Laplacian application are the same in sequential execution. The results from parallel execution show a speed-up of only 3.2. The discrepancy between the results of standard OpenMP development and the ΣC runtime generation comes from the gain of pipeline parallelism. Indeed, building an efficient pipeline with OpenMP is quite difficult, so the data-flow runtime gains an edge from this added parallelism.

5 Related work

Research work on data-flow paradigms and execution is done very extensively, because it is, as noted in the introduction, a programming model which would fit well the future of parallel architectures. Nonetheless, when we focus on compilers for data-flow languages, we restrain the number of related work significantly.

In addition to the StreamIt [2] and Brooks [4] languages (whose development has been stopped) that we already cited, and if we put aside the synchronous languages like Esterel since they do not really target manycore systems, only a handful of pieces of work remains.

In the HPC domain, there is an ongoing effort at BSC, around an OpenMP extension for Data-flow called OmpSs [1]. The noticeable differences are that OpmSs is a simple C extension using pragmas (alike OpenMP), and mostly targeted toward HPC, whereas ΣC is initially targeted at High Performance Embedded Systems (HPES).

Another significant piece of work is OpenStream [13] which, like OpenMP, is an extension of C using pragma, but targeting also embedded systems. We expect the performance of ΣC to be better, especially on constrained embedded systems, because of the scale of development of ΣC compared to OpenStream. Nonetheless, both share a lot of similarities.

6 Further advantage and conclusion

One of the advantage of having an OpenMP backend for ΣC, is that we can use the OpenMP primitive inside any given agen. This can be used to increase easily the parallelism ratio of an application. We used this trick to parallelize the image transposition agent in the Laplacian application, and we tested the result on a AMD Opteron 6172 system with a total of 48 cores. The results is shown in Figure 4.

As can be seen, this system is much less efficient than the Xeon Workstation, as even 8 full threads only give a speed-up of 2.72 instead of 3.7 for the 4 core Xeon, without pipelining. Using the trick of mixing ΣC and OpenMP results in increased performance with a top speed-up of 4.2 to be compared to 2.82 for the initial pure ΣC, which represents an improvement of nearly 50%. When pipelining is enabled, the speed-up is much more impressive, even overlinear, probably because the program is memory-bound.

For the moment such trick only works with the OpenMP backend of ΣC, since it is not taken into account by the ΣC compiler but by the final C compiler. We could imagine using

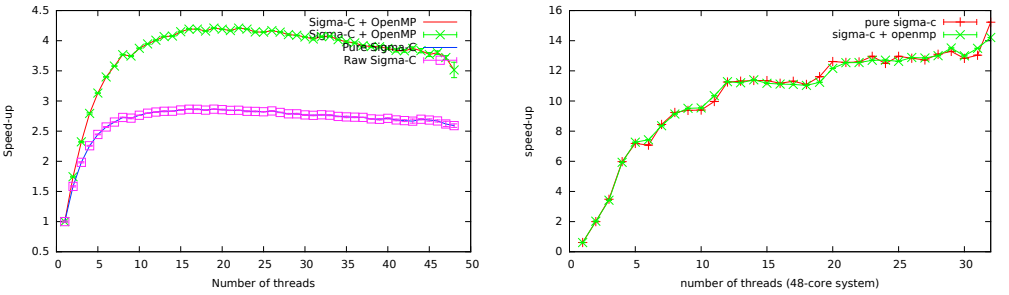

Figure 4: Speed-up comparison on the pure ΣC results speed up and ΣC with an added OpenMP parallel loop in agent Transpose, for a 48 core system (left without pipelining, right with pipelining enabled)

a subpart of OpenMP to include it in the ΣC compiler, if we can find a way to transform the OpenMP code to CSFD. This would obviously work only for a well chosen subset of OpenMP but seems an interesting approach. This is future work.

References

[1] Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[2] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2/3):261–278, Jun 2005.

[3] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In V. N. Alexandrov, M. Lees, V. V. Krzhizhanovskaya, J. Dongarra, and P. M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1624–1633. Elsevier, 2013.

[4] I. Buck. Brook specification v.0.2. Technical report, Stanford University, 2004.

[5] L. Cudennec and R. Sirdey. Parallelism reduction based on pattern substitution in dataflow oriented programming languages. *Procedia Computer Science*, 9(0):146 – 155, 2012.

[6] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Leger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Propedia Computer Science, International Conference on Computational Science (ICCS-2013), Alchemy Workshop*, 18, 2013.

[7] P. de Oliveira Castro, S. Louise, and D. Barthou. Reducing memory requirements of stream programs by graph transformations. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 171 –180, 28 2010-july 2 2010.

[8] X.-K. Do, S. Louise, and A. Cohen. Comparing the StreamIt and ΣC languages for manycore processors. In *Fourth International workshop on Data-Flow Models for extreme scale computing (DFM 2014, associated with PACT)*. ACM, 2014.

[9] R. L. G. Bilsen, M. Engels and J. A. Peperstraete. Cyclo-static data flow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[10] P. Guillou. *Compilation efficace d'applications de traitement d'images pour processeurs manycore*. PhD thesis, École nationale supérieure des mines (Paris), Nov. 2016.

[11] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[12] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[13] A. Pop and A. Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, Jan. 2013.