ISSN 0005-1144
ATKAFF 51(4), 387–396(2010)

Josip Knezović, Mario Kovač, Hrvoje Mlinarić

Integrating Streaming Computations for Efficient Execution on Novel Multicore Architectures

UDK 004.272.43.042
IFAC 2.8.1

Original scientific paper

Streaming has emerged as an important model in present–day applications, ranging from multimedia to scientific computing. Moreover, the emergence of new multicore architectures has resulted with new challenges in efficient utilization of available computational resources. Streaming model offers the portability and scalability of performance with the increasing number of cores. In this paper we propose a tool which enables the implementation of the compute–intensive stream processing kernels as portable modules in general–purpose applications. Resulting modules can be efficiently reused with high degree of scalability in regard to increasing number of processing cores.

Key words: Streaming model, Parallel programming, Signal processing, Multicore

Integracija tokovnog modela za učinkovito izvođenje na višejezgrenim računalnim arhitekturama. Tokovni računalni model predstavlja zanimljivo područje istraživanja s ciljem ubrzanja kako multimedijskih, tako i znanstvenih aplikacija. Isto tako, pojava višejezgrenih računalnih arhitektura rezultirala je povećanjem zanimanja za istraživanje metoda i modela koji bi omogućili učinkovito iskorištavanje postojećih paralelnih resursa. Tokovni model omogućuje istovremeno visok stupanj apstrakcije, prenosivost i skalabinost aplikacija s obzirom na povećanje računskih jezgri. U ovom je članku predložen pristup koji omogućuje implementaciju računski zahtjevnih dijelova aplikacija u tokovnom modelu te njihovu integraciju u vidu prenosivih modula. Na taj način ostvareno je ubrzanje cjelokupnih aplikacija pri izvođenju na višejezgrenim procesorima.

Ključne riječi: tokovni računalni model, paralelni sustavi, obrada signala, višejezgrena računala

1 INTRODUCTION

Reaching the point of diminishing returns in increasing the clock frequency of uniprocessors has led to the emergence of novel multicore architectures. We have seen the proliferation of homogeneous and heterogeneous multiprocessors in not only general–purpose computing, but in embedded computing as well. Examples are Cell BE [1], Tilera's TILE64 [2], Sun's Niagara and Rock [3], Intel Core 2 Duo, Quad, Teraflops [4], AMD's Barcelona [3] etc. Therefore, multiprocessor technology is no longer reserved for high–performance scientific and special purpose applications only, but to the ubiquitous world of general purpose and embedded computing as well. Available solutions range from symmetric multiprocessors with shared memory to asymmetric multiprocessing systems with or without shared memory demonstrating wide diversity of possible computing platforms. In order to exploit the abundance of the computing resources, the trend must be accompanied with appropriate shift in computational models, programming language and compiler technology which should provide easy, efficient and portable implementation of parallel programs. Moreover, many existing and upcoming applications contain compute–intensive data centric computations such as image, video and audio coding, playback and processing, gaming, multimedia content processing, retrieval and archival. They all have streaming nature due to stream–like processing of large amount of data. These applications can therefore substantially benefit from upcoming multicore architectures by exploiting the available parallelism and executing compute–intensive parts in parallel.

The main idea of the streaming processing model is that the program is expressed as a set of independent and infrequently changing operations on the huge amount of data. Data items are processed and transported among processing elements connected with the communication channels. Numerous past and recent research projects such as Imagine [5], StreamIt [6], RAW [7], Merrimac [8], Brook [9] have exploited the idea of streaming. Recently, emerging multicore architectures have motivated a renew in the research for parallelization of computations expressed in the streaming model. The results show the promising results in

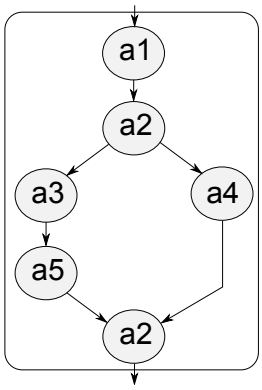

Fig. 1. An example of a stream program graph

terms of providing the platform for efficient and portable implementation of data–intensive applications for parallel processing.

In this paper we propose a methodology and a tool which enables the implementation of compute–intensive stream processing kernels of contemporary data processing applications as streaming computations. We express these kernels in the StreamIt, a high–level stream programming language, and integrate and reuse them in a typical large–scale data–intensive application. By expressing them in such high–level, domain–specific model we obtain application portability and scalability of performance with the increase of cores in a multicore processor. We show the benefits of our approach by evaluating the performance of several popular stream processing kernels. We also demonstrate the scalability of our approach with the increasing number of cores.

The rest of the paper is organized as follows: Section 2 gives a brief overview of the stream programming paradigm and introduces the StreamIt, a research–based streaming language characterized with the platform–independence and high abstraction level. In section 3 we present our ideas regarding the separation of streaming kernels and their implementation and integration into general–purpose applications. The kernels are expressed in StreamIt which enables the portability and scalability of performance with the increase of cores on a multicore processor. Finally, in section 4 we demonstrate the experimental results that we obtained with our proposal. Section 5 gives conclusions and summarizes our ideas with regard to future challenges and plans.

2 BACKGROUND

2.1 Stream Programming

Stream programming paradigm offers a promising approach in exposing the parallel resources of multicore architectures to data–intensive and computationally demanding applications. Often, many parts of these applications can be expressed as the set of independent transformations on the data they process, thus exhibiting the streaming nature. Recently, there has been an increase in the interest in the model which resulted in several popular streaming languages such as StreamIt [10], Brook [9], Spidle [11], and CUDA [12]. This was motivated not only by trends in multicore architectures, but also by trends in the applications with the emergence of image, video, network and multimedia processing or, more generally, data–intensive processing. One of the benefits of stream programming paradigm is the clean separation between computations and the communication carried on the data. A program is expressed as a structured graph of actors, also referred to as processing kernels or filters, and communication channels. Data items are presented as streams of tokens flowing through the graph [13]. An illustrating example of a stream graph is shown in Fig. 1.

The separation of processing elements and communication channels is an important aspect of the streaming computational model. Each actor takes a fixed number of input elements, i.e. tokens from its input stream, processes them and outputs a fixed number of results to its output stream. Since the actors can only access its input and output tokens, they can be viewed as isolated computations having their own program counter and private and independent address space, thus making all the memory accesses localized. These properties enable the streaming compiler to efficiently analyze and parallelize the stream program for the target parallel machine.

Actors have firing rules, ie. they execute and therefore process input data when certain conditions are satisfied, such as the number of input tokens on the input channel. The firings of actors is scheduled in a periodic manner taking their input requirements into account. Since all actors have their fixed memory space and well defined work functions, all dependencies between actors are made explicit by communication channels. This enables the explicit parallelism between actors to be easily expressed, thus leveraging the compiler to efficiently parallelize and optimize the whole program [14]. Stream graph differs from traditional, sequential program flow graph in that all of the actors of the graph are implicitly running in parallel and their execution order is constrained only by the availability of data on their input channels. The model asserts the independence of the actors since they only communicate with their immediate neighbors. This eliminates the existence of any non–local dependencies of one actor to another and enables the compiler to efficiently orchestrate the execution of the graph. However, due to the vast diversity in emerging architectures, providing an efficient mapping of stream graph to multicore platform presents a challenging task. Nevertheless, stream programming concepts allow

the programmer not to worry about hardware specifics, but to concentrate on the problem and to express its parallelism in abstract and portable manner. The task of the streaming compiler is to efficiently map stream graph onto the target multicore architecture.

In a streaming domain, programmer has the ability to express different types of parallelism, ranging from the fine grain data parallelism to coarse grain task parallelism. Data parallelism, expressed easily as an actor that has no interdependence on the input data between its any two executions, is implicitly stated. It offers unlimited amount of parallelism that can be exploited by the streaming compiler since the data–parallel actors can be spread across any number of computational units, depending on the architecture of the platform and on the availability of input data. Other types of parallelism such as task parallelism and pipeline parallelism should be exploited as well if efficient utilization of processing cores is required. Task parallelism in streaming domain is expressed as independent, parallel branches in the stream graph such as actors *a3* and *a4* in Fig. 1. Their execution is separated since the output of the actor *a3* never reaches the input of actor *a4* and vice versa. Task parallelism reflects the logical, algorithmic parallelism [14], and its amount in stream programs depends of the application and on the programmer's view. Pipeline parallelism applies to chain of directly connected actors in a stream graph. It is similar to hardware pipelining, but it is carried on the higher level between actors that have producer–consumer relationship in the stream graph. It offers the gains by running the stages of the pipelined executions in parallel with the drawback of increasing the latency and buffering between pipelined actors [7].

Despite the abundance of parallelism in streaming applications, the synchronization overhead associated with the communication in the stream graph presents a challenge to the overall system performance. This is affected by the granularity of actors or more specifically, their algorithmic intensity. If the actor process time is low compared to the time spent to prepare input and output data, communication cost will overshadow the computation benefits obtained through the parallelization. In contemporary shared–memory multicores, memory bandwidth is substantially smaller than the processing bandwidth, which greatly affects their scalability. These effects are efficiently dealt with by the streaming compiler which performs a number of stream graph transformations in order to produce efficient communication pattern and load balancing of the graph, something which is not possible in traditional programming models.

2.2 StreamIt Programming Language

StreamIt is a platform–independent and high–level programming language aimed to modern stream processing [15]. It is based on structured representation of stream graph enabling the program organization as a hierarchical graph of processing nodes. The language enables robustness and expressiveness of the parallelism together with the portability and scalability across various multicore architectures. Additionally, it provides the compiler infrastructure for stream specific optimizations on a stream program mapping to underlying processor architecture.

```
int->int filter Decimator(int n) {
  work pop n push 1 {
    for (int i = 0; i < (n-1); i++)
      pop();
    int x = pop();
    push(x);
  }
}
```

Fig. 2. An Example of a StreamIt filter

In StreamIt, the basic programmable computational unit is referred to as the *filter*. An example of a filter definition is shown in Fig. 2. All communication is carried on the filter's input and output tape, also known as channels. Occasionally, irregular data are communicated via control messages also known as teleport messages, if necessary [16]. Filter specifies its input and output type, ie. type of data it consumes and produces, which doesn't have to be primitive only, but user–defined type as well. Basic unit of computation is the *work* function that repeatedly reads data from the input channel, processes them and outputs the results on the output channel. The input and output tapes are organized as FIFO channels from which filter *pops*, or *push*es data. Additionally, filter can also *peek* data on its input without removing it. The number of data consumed, produced and peeked for one invocation of the *work* function is specified by the filter pop, push and peek rate. The rates are usually statically declared, which enables the StreamIt compiler to apply various optimizations and construct efficient execution schedules for target multicore architecture [17]. Filters can also declare its own private variables or maintain the state between its executions. If the state is required, the filter is referred to as *stateful*, otherwise the filter is *stateless* [16]. Aside to programmable filters, StreamIt defines several built–in filters that perform specific operations such as file input, output and data forwarding. They are shown in Table 1.

A complete stream program is described as a hierarchical graph of filters, either programmable or built–in. Hierarchical structures used to construct the graph are: *pipeline*, *splitjoin* and *feedback loop*, as shown in Fig. 3 [17]. Pipeline is a single input to single output parameterized stream. It is composed as a sequence of

Table 1. StreamIt built–in (native) filters

Filter	Description
`Identity<type>`	Forwarding filter. Takes an input element of specified `type` from the input tape and pushes it on the output tape.
`FileReader<type>`	Parameterized file input. Performs reading of the data of specified `type` from the file and pushes them to its output tape. Generally used at the beginning of the StreamIt program.
`FileWriter<type>`	Parameterized file output. Performs writing of the data of specified `type` to the file by taking it form its output tape. Generally used as the last filter in the StreamIt program.

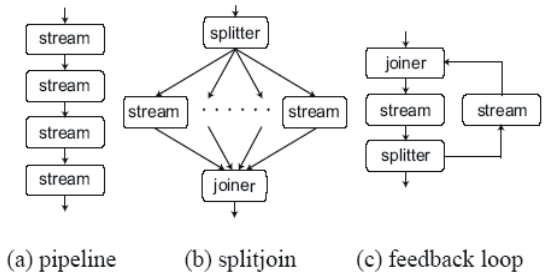

Fig. 3. StreamIt hierarchical streams

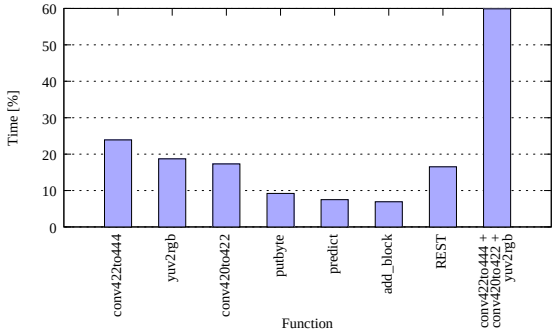

Fig. 4. Execution time profile of an MPEG decoder

stream nodes, either filters or other hierarchical streams. In a pipeline one filter sends its output to the input of the filter that follows in the pipeline. Splitjoin is a construct that allows distribution of a data to the set of parallel streams. Those streams are joined back in a roundrobin fashion. Splitter scatters the data, and joiner gathers back. This allows the programmer to express task parallelism in applications. Feedback loop introduces cycles in the stream graphs. Although possible, this construct is not very often used as it can always be expressed as a stateful filter.

Although it leverages the synchronous dataflow SDF model of computation [18], StreamIt extends it with the support for dynamic rates of filters, peeking on the input tapes, and timed control messages through the concept referred to as teleport messaging. These extensions are crucial to the broader use of streaming model of computation, ranging from general purpose applications to data centric and scientific applications.

3 INTEGRATION OF STREAMING MODULES FOR EFFICIENT EXECUTION

Past experience in implementing complex applications which contain compute–intensive streaming parts such as MPEG–2 decoder has shown the difficulties in implementing them entirely in the streaming domain [19]. This is rather true for any domain–specific computational model due to the fact that in the real–life applications there exist parts of the code which pertain to the handling of user inputs, options, irregular events, termination conditions, fault handling etc. This type of program logic is very hard to implement in streaming domain. Moreover, stream–oriented parts of these applications such as DCT, FFT, image segmentation, are usually the most processor–hungry parts. This effect is clearly visible in Fig. 4 where an execution profile of a MPEG–2 decoder is shown. The benchmark we used was the reference decoder implementation and a part of the MediaBench benchmark suite [20]. First three bars in the figure represent the contribution of pixel upsampling functions `conv420to422` and `conv422to444` and color conversion `yuv2rgb` in the overall decoder execution. These parts together constitute almost 60 percent of the execution time. Clearly, they exhibit streaming nature and are naturally expressed in the streaming model. Other parts of the decoder are devoted to program logic which is moderately or very hard to implement in the model. In addition, isolated functions are natural target for eventual performance–oriented optimizations. By implementing them in the streaming domain, we strive to achieve two important goals: high–level portability among different architectures and the scalability of performance with the increased number of processing cores.

As we observed that the design of real–world data–intensive applications is hard to fit entirely in a domain–specific model such as streaming, we propose to take an-

other approach in which the streaming parts are designed within the streaming domain, while other parts of the application are left outside the domain and designed using traditional programming languages. In case of the existing application targeted for efficient execution on the multicore, we isolate the streaming kernels which are usually computationally the most demanding parts, usually by profiling it. When identified, streaming parts will be implemented in streaming model and integrated into the host application.

In order to explore the potential benefits of our proposal we implemented the design flow and the toolchain illustrated in Fig. 5. We modified the StreamIt compiler chain (dashed blocks were expunged) in order to produce streaming computations as routines which can be integrated into the host application code. The application is divided into parts which are implemented in StreamIt and the general parts which are implemented or left in the C/C++ programming language. StreamIt compiler is a high–level source–to–source compiler that takes a StreamIt source file and generates C/C++ code linked against the underlying threading library such as PThreads. The compiler assumes that the complete program is expressed in StreamIt. Therefore, in order to implement our idea, we had to modify the compiler so that the code for `FileReader` and `FileWriter` filters is removed. Additionally, resulting C/C++ code is modified so that it could be linked against the application source code, and the code that performs the exchange of the data between streaming kernel and the application code is inserted. The data exchange is implemented through dynamically allocated buffers in the memory, which is done in the application code. Additionally, StreamIt compiler generated the `main` routine for the streaming part which is also replaced with the entry routine for the application code to call streaming computations. The rest of the compiled StreamIt program, i.e. stream code which contains the filters that perform operations were unchanged. Finally, the application program code and streaming code are compiled by a native compiler such as gcc resulting in an executable program.

The design flow showed here serves for the purpose of proving the ideas of our proposal. Our current efforts are the addition of built–in filters into StreamIt which will enable more flexible exchange of data and with the more automated toolchain. We also plan to provide more flexible, low–overhead streaming runtime which will enable the invocation of compute-intensive streaming computations, something similar to foreign language runtimes available for other domain specific languages such as Sisal [21]. In addition, we are currently investigating the possibilities to modify the StreamIt language to add constructs and intrinsic filters for efficient multidimensional data exchange and processing.

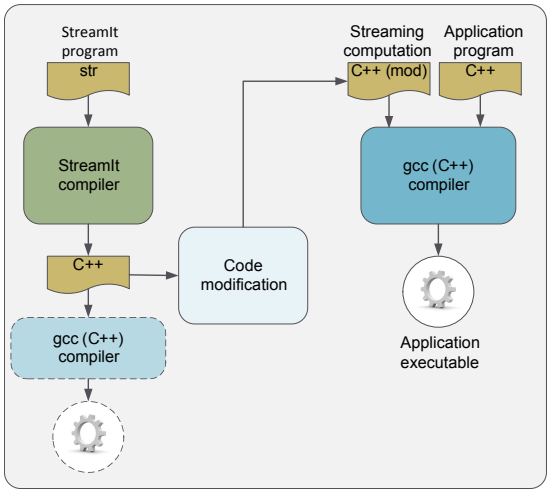

Fig. 5. The design flow of proposed methodology

4 EXPERIMENTS AND RESULTS

In order to validate our proposal we investigated it on a set of benchmarks described in Table 2. These programs, except for CBP, are extracted from the versatile StreamIt benchmark suite available at [22]. In our experiments, we modified each program so that it consists of the main application part (the host) programmed in C/C++ and streaming kernel programmed in StreamIt, as illustrated in Fig. 5. The kernel was compiled with StreamIt compiler that we have modified in order to implement our proposal. Generated C code for the kernel was finally interfaced and integrated into the host application. As already noted, the host application code usually performs the general program logic such as error handling and user options handling. Additionally, it handles data input from files or some other source. Data items are sent to the streaming kernel which is invoked in order to perform its computations. After processed by the kernel, the results are gathered and sent back to the host application. As previously said, this approach enables the compute–intensive streaming computations to be expressed in a domain–specific model which will provide portability and scalability over the wide range of multicore architectures, in terms of both, architectural changes and increasing number of cores.

Figure 6 shows the execution time breakdown for selected benchmarks. We show the contribution of the time spent in the streaming kernel and in the rest of the benchmark (the host) as percentage of the total execution time. As can be seen, streaming parts dominate the execution time, which makes them the natural target for performance optimization. Our benchmarks have only one streaming kernel implemented in StreamIt, while the rest of the program is implemented in C or C++ as the host. In future, we plan to add support for integration of more streaming

Table 2. Evaluated benchmarks

Benchmark	Filters	Description
Fmradio	65	Software FM radio with equalizer
Filterbank	51	Filter bank for multirate signal processing
FFT	97	Fast Fourier transform
DCT	38	2D DCT transform 8x8
CBP	44	Contextual pixel predictor for lossless coding
AudioBeam	21	Audio beamformer for recording with a large microphone array
RGB2YUV	7	Color space conversion

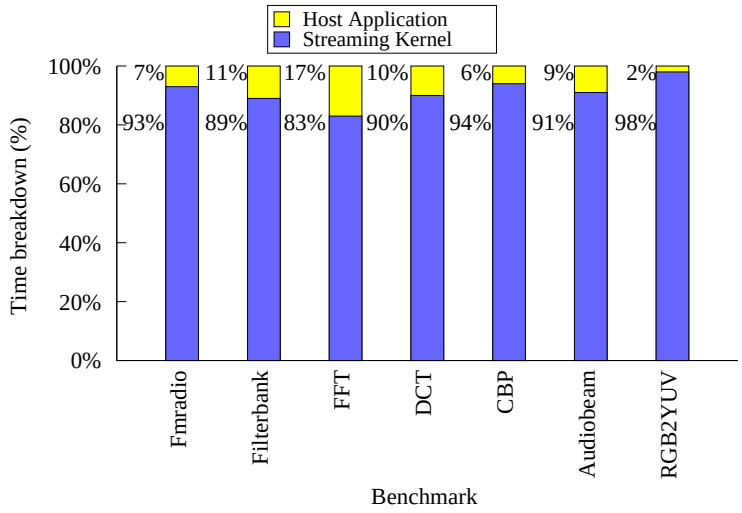

Fig. 6. Execution time breakdown of selected benchmarks

kernels into the host and for efficient thread management for the streaming runtime.

Illustrating examples for the case of *audiobeam* benchmark are shown in Figs. 7 and 8. Figure 7 shows the original stream graph of the *audiobeam* streaming kernel, as perceived by the programmer, while Fig. 8 shows the overall diagram of the final program as generated by our tool, consisting of the host application code and compiled code for the *audiobeam* streaming kernel. Our tool replaces the StreamIt `FileReader` and `FileWriter` filters with the data exchange code which is inserted in the generated streaming code and the host code. Original stream graph from the Fig. 7 is a fine grained, algorithmic representation of the kernel. Every branch processes the data collected by one microphone, the figure shows the setup with eight microphones, actual number of microphones used for experimental evaluation was 15. The programmer does not have to worry about the load balancing of streaming nodes and other architecture–specific details of target machine. The role of the streaming compiler is to partition the stream graph into a set of well balanced nodes with a set of available transformations with the constraint of preserving functional equivalence of the original and final stream graph. The effect of streaming transformations are shown in Fig. 8 where the resulting stream graph is shown targeted to the dual core Pentium D machine. The partitioned streaming module resulted in two parallel threads represented by two nodes in the middle of the figure. The data items for threads are scattered with the roundrobin splitter and gathered again by roundrobin joiner before sent back to the main application program. The resulting code generated by the streaming compiler and application code are finally compiled and linked together into the resulting binary by the *gcc* compiler.

We ran our experiments on two machines resembling the trend of increasing the number of cores: a 2.66 GHz dual core Pentium D with 1 GB of memory (*2 Cores*) and a 2.66 GHz quad core Core 2 Quad processor with 4 GB of memory (*4 Cores*). Both systems ran a Linux operating system with kernel version 2.6.23. Resulting program code in C/C++ consisting of the main application code and the streaming code generated by the StreamIt compiler, was compiled into executable by the `gcc` version v3.4 with the –O3 optimization set. StreamIt compiler was instructed with no stream–specific optimizations, the only option was the target number of threads. For

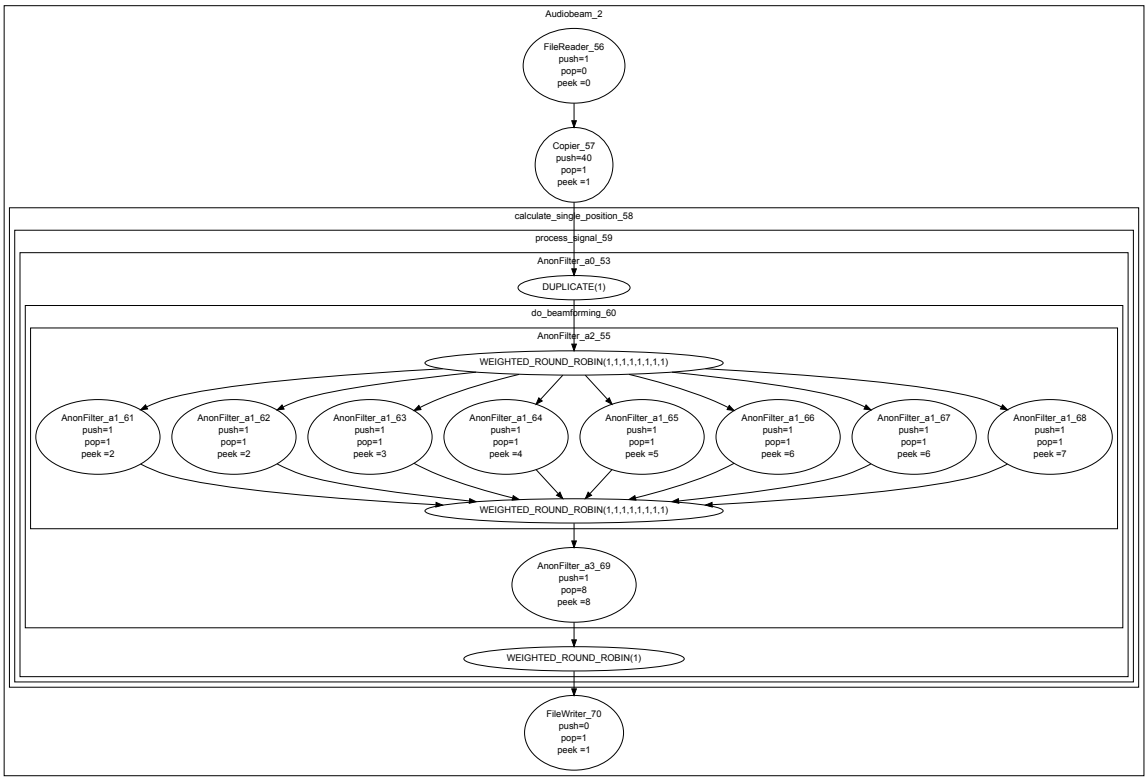

Fig. 7. Audiobeam benchmark: Original stream graph

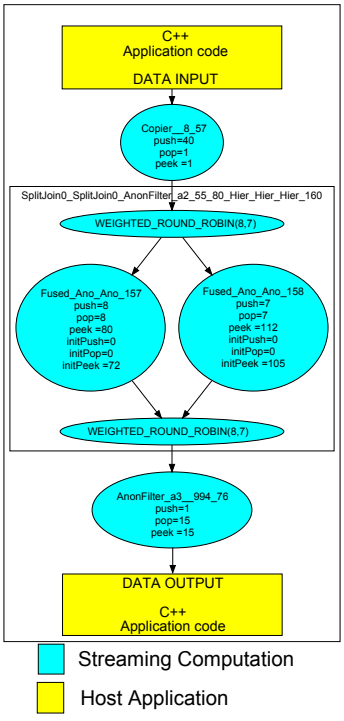

Fig. 8. Audiobeam: Resulting graph for two core machine

every machine we measured the speedup of the program that parallelizes the streaming part by spawning it to multiple threads over the single threaded version for which the streaming computations utilize one thread of execution. The speedup was computed as the ratio of the response time t_s of the single threaded version to t_m the response time of the multithreaded version of the program on the same machine. Multithreaded version was obtained by selecting the target number of threads for the streaming kernel which resulted in the best execution time of the overall program.

A summary of our results appears in Table 3 and Fig. 9 where we show obtained speedups using our tool. For each program we measured the speedup obtained on the running systems (*2 Cores* and *4 Cores*) and the theoretical speedup bounded by the Amdahl's law and execution profile in Fig. 6 (*2 Cores Max* and *4 Cores Max*). Our approach yielded linear speedup improvements in execution time compared to single threaded version for all benchmarks except FFT. We find reason for misbehavior in this case because of the wildly unbalanced stream graph of the stream graph with the highly unmatched rates of neighboring filters resulting in significant blocking during the execution, which was previously reported [7]. In case of the CBP benchmark, we obtained super–linear speedup of

Table 3. Obtained speedups

Benchmark	2 Cores	2 Cores Max	4 Cores	4 Cores Max
Fmradio	1.82	1.87	3.21	3.31
Filterbank	1.68	1.80	2.53	3.01
FFT	1.43	1.71	1.21	2.65
DCT	1.77	1.82	2.98	3.08
CBP	2.22	1.89	3.45	3.39
Audiobeam	1.73	1.83	3.11	3.15
RGB2YUV	1.92	1.96	3.72	3.77
Geom. Mean	**1.78**	**1.84**	**2.74**	**3.18**

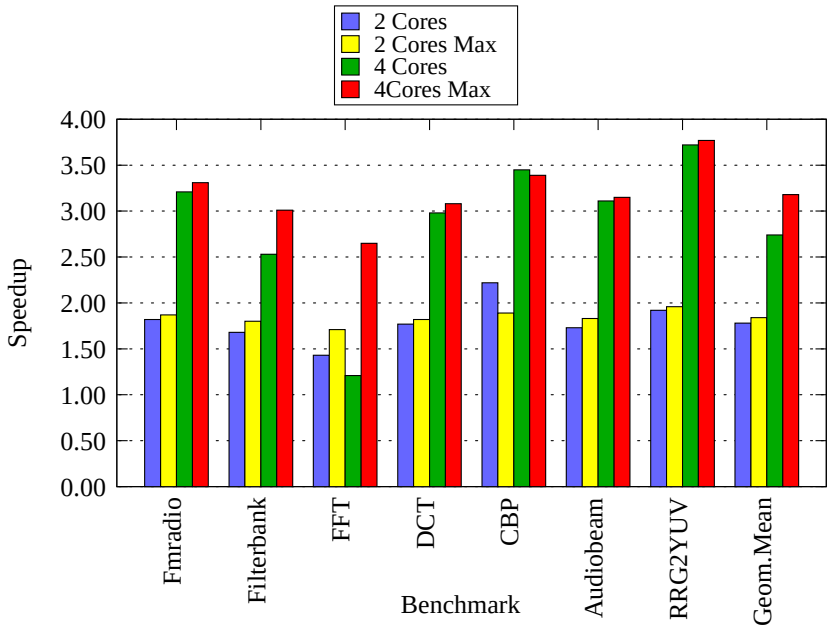

Fig. 9. Performance results

2.22 for a two core machine, and 3.45 on a four core machine. We attribute this to the cache and data locality effects which were substantially improved for the resulting stream graph compared to the original, fine–grained stream graph with the nodes exhibiting low arithmetic intensity. Geometric mean speedup of 1.78 for dual core system and 2.74 for quad core system clearly demonstrates the benefits of our proposal.

5 CONCLUSIONS

Stream processing kernels exist in many present–day applications such as video playback, gaming, multimedia etc. Traditional, imperative programming languages can not provide an efficient platform and required level of abstraction required for portable and scalable implementation of these computations on novel multicore architectures. Moreover, their parallel nature can not be efficiently expressed in such sequential model, leaving the programmer to trade the performance for portability. On the other hand, streaming computational model provides a natural way to express streaming kernels as parallel computations by exposing the data, pipeline and task parallelism. However, real world applications are not easily expressed as stream programs because of a lot of logic devoted to other non–stream oriented tasks such as user input handling, error checking etc. Because of that, we proposed an approach where data intensive streaming parts of contemporary applications are expressed in streaming domain and integrated into the host application as reusable modules. In this paper we demonstrate promising initial results in using this approach in terms of both: the portability across different computer architectures and scalability with increasing number of cores.

Although our proposal showed promising results, there

remains lot of space for improvements in terms of both programmability and efficiency. We plan to introduce better support in StreamIt by providing the language with the constructs and built–in filters which will enable one–dimensional and multidimensional data exchange between streaming modules and the host application. Moreover, the exchange of user–defined types aside from intrinsic types is planned. We would also like to introduce the streaming computations as part of the domain–specific foreign–language interface and runtime which will manage parallel resources of underlying machine more efficiently, similar to interfaces in [23–25]. This approach should bring most benefits in enabling efficient thread creation and reuse inside the streaming runtime as well as more flexible usage of the streaming model of computation and more parallelization opportunities by implementing the high–level pipelining of the streaming computations and the host application.

Although demonstrated on commodity multicores with shared memory, our proposal could be extended so that the streaming computations are executed on other computing platforms for which there exists StreamIt compiler support such as GPUs [26], heterogeneous multiprocessors [27] and reconfigurable logic [28], thus extending the portability of our approach.

REFERENCES

[1] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589–604, July–September 2005.

[2] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On–Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15–31, September/October 2007.

[3] J. McGregor, "The new x86 landscape," *Microprocessor Report*, May 14 2007.

[4] K. P, A. K, and O. K, "Niagara: a 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, pp. 21–29, March/April 2005.

[5] S. Chatterji, M. Narayanan, J. Duell, and L. Oliker, "Performance evaluation of two emerging media processors: VIRAM and Imagine," in *Proc. Parallel and Distributed Processing Symposium 2003*, April 2003.

[6] W. Thies, M. Karczmarek, M. Gordon, D. Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe, "StreamIt: A compiler for streaming applications," Tech. Rep. Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, Dec. 2001.

[7] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," in *Proc. ASPLOS 02 International Conference on Architectural Support for Programming Languages and Operating Systems*.

[8] J. Gummaraju, M. Erez, J. Coburn, and M. R. W. J. Dally, "Architectural Support for the Stream Execution Model on General-Purpose Processors," in *Proceedings of the 16th Int'l Conference on Parallel Architectures and Compilation Techniques PACT 07*, (Brasov, Romania), September 2007.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777–786, August 2004.

[10] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs," in *International Symposium on Microarchitecture*, (Chicago, IL), Dec 2007.

[11] C. Consel, H. Hamdi, L. Réveillière, L. Singaravelu, H. Yu, and C. Pu, "Spidle: a DSL approach to specifying streaming applications," in *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, (New York, NY, USA), pp. 1–17, Springer-Verlag New York, Inc., 2003.

[12] J. Nickolls and I. Buck, "CUDA Software and GPU Parallel Computing Architecture," *Microprocessor Forum*, May 2007.

[13] J. Gummaraju and M. Rosenblum, "Stream Programming on General-Purpose Processors," in *MICRO 38: Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, (Barcelona, Spain), November 2005.

[14] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), October 2006.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreaMIT: A language for streaming applications," MIT/LCS Technical Memo LCS-TM-620, Massachusetts Institute of Technology, Cambridge, MA, Aug. 2001.

[16] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, "Teleport messaging for distributed stream programs," in *Symposium on Principles and Practice of Parallel Programming*, (Chicago, Illinois), Jun 2005.

[17] M. Drake, D. Zhang, M. Gordon, J. Sermulins, W. Thies, A. Dimock, R. Rabbah, and S. Amarasinghe, "Ubiquitous stream programming to facilitate the migration to multicore architectures," in *Proc. STMCS First Workshop on Software Tools for Multi-Core Systems*, March 2006.

[18] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[19] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," in *Proc. International Parallel and Distributed Processing Symposium*, (Rhodes Island, Greece), 2006.

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[21] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller, *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, ch. The Sisal Project: Real World Functional Programming, pp. 45–72. Springer-Verlag New York, USA, 2001.

[22] StreamIt Benchmarks, "http://www.cag.csail.mit.edu/streamit/shtml/benchmarks.shtml," June 2009.

[23] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Miller, "The Sisal Model of Functional Programming and its Implementation," in *Proceedings of pAs'97, Aizu-Wakamatsu, Japan*, March 1997.

[24] N. Marcussen-Wulff and S.-B. Scholz, "On interfacing SAC modules with C programs," in *Proceedings of the 12th International Workshop on the Implementation of Functional Languages (IFL 2000)* (M. Mohnen, ed.), pp. 381–386, RWTH Aachen, 2000.

[25] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," in *Proceedings Euro–Par 2009 (To appear)*, August 2009.

[26] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software Pipelined Execution of Stream Programs on GPUs," in *2009 International Symposium on Code Generation and Optimization (CGO)*, (Seattle, WA), March 2009.

[27] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. ACM SIGPLAN 2008 Conference on Programming Languages Design and Implementation (PLDI)*, pp. 114–124, ACM, June 2008.

[28] A. Hormati, D. B. Manjunath Kudlur, S. Mahlke, and R. Rabbah, "Optimus: Efficient Realization of Streaming Applications on FPGAs," in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)2008*, (Atlanta), October 2008.

Josip Knezović received his B.Sc., M.Sc. and Ph.D. degree in Computer Science from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2001, 2005 and 2009, respectively. Since 2001 he has been affiliated with Faculty of Electrical Engineering and Computing as a research assistant at the Department of Control and Computer Engineering. His research interests include programming models for parallel systems in multimedia, image and signal processing. He is the member of IEEE and ACM.

Mario Kovač is full professor at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. He is also an executive, author and expert in the field of multimedia hardware and software design, business models for computer systems, chip architecture design and mobile systems. He is serving as a consultant in the area of multimedia systems and applications and business models for large computer systems. In 1993–94, Dr. Kovač was given a Fulbright scholarship for computer science and engineering research that he spent in the USA. Professor Kovač served as Head of the Dept. of Control and Computer Engineering and Vice Dean for Business Development at the Faculty of Electrical and Computer Engineering. Currently he is mostly involved in business relations with industry partners. He is President of the Management Board of Croatian Academic and Research Network and VP of the Supervisory Board at BICRO. Prof. Kovač was awarded with the Medal of Honor by the President of the Republic of Croatia: Order of Croatian Danica with the Image of Ruđer Bošković in 2008. He has several U.S. patents for multimedia and SC related technologies. He is senior member of the IEEE Computer Society, the Croatian Academy of Engineering and several other societies.

Hrvoje Mlinarić received his B.Sc., M.Sc. and Ph.D. degree in Computer Science from the Faculty of Electrical Engineering and Computing, University of Zagreb in 1996, 2002 and 2006, respectively. Since 1997 he has been with Faculty of Electrical Engineering and Computing currently holding the assistant professorship position. He is also the vice-head of the Department of Control and Computer Engineering. His research interests include data compression, programmable logic and advanced hardware and software design. He is the member of Croatian Academy of Engineering.

AUTHORS' ADDRESSES

Josip Knezović, Ph.D.
Prof. Mario Kovač, Ph.D.
Asst. Prof. Hrvoje Mlinarić, Ph.D.
Department of Control and Computer Engineering,
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000 Zagreb, Croatia
email: josip.knezovic@fer.hr, mario.kovac@fer.hr,
hrvoje.mlinaric@fer.hr

Received: 2010-11-03
Accepted: 2011-01-28