Tackling Real-Time Signal Processing Applications on Shared Memory Multicore Architectures Using XPU

Nader Khammassi [§, £]

[§] Radar & Warfare Systems
Thales Airborne Systems
29200 Brest , France

nader.khammassi@ensta-bretagne.fr

Jean-Christophe Le Lann [£]

[£] Lab-STICC UMR CNRS 6285
ENSTA-Bretagne
29806 Brest Cedex 9, France

jean-christophe.le_lann@ensta-bretagne.fr

***Abstract*—General-purpose shared memory multicore architectures are becoming widely available. They are likely to stand as attractive alternatives to more specialized processing architectures such as FPGA and DSP-based platforms to perform real-time digital signal processing. In this paper, we show how we can ease parallelism expression on shared memory multicore architecture through the XPU high-level programming model and we describe a parallel implementation of radar signal processing application. This study case shows how we can improve programmer productivity through easing parallel programming without sacrificing performances.**

***Keywords*- Parallel Parallel Programming Model, Skeleton, Parallel Construct, Patterns, Pipeline Parallelism, Multicore, Digital Signal Processing, Radar**

I. Introduction

While parallel programming is still a hard task for the average sequential programmer, the continuous proliferation of parallel hardware has placed developers under great pressure to parallelize their applications in order to take advantage of these platforms [1,2,3,4,5]. Moreover, the use of specialized parallel processing devices such as FPGA and DSP may make parallel programming even a harder task since it requires a deep understanding of the target hardware architecture in addition to strong parallel programming knowledge and skills. Such hardware may offer great performance but at the cost of poor programmability and consequently low productivity. Recent general-purpose multicores architectures (GPMA) [2], such as the widely available x86 architectures, can offer an attractive alternative to those specialized devices since they can achieve acceptable performances at the cost of less programming effort. Particularly, stringent real time requirements for most digital signal processing applications seem still beyond reach of today GPMA: these DSP applications require supplemental dedicated accelerators or even fully dedicated SoC architectures. However, GPMA show such promises and it is likely that in a near future, DSP applications could be modeled appropriately and executed in such GPMA.

Many parallel programming models have been designed for GPMA. These programming models have different approaches and provide different trade-off between productivity and performance [1,19]. Low-level programming model such as “pthread” may achieve high performance but suffers from poor programmability. At the opposite side, many high-level programming models tend to sacrifice performances to offer good programmability and to improve consequently programmer productivity. In the middle, many other high-level programming models make various productivity-performance tradeoff. Particularly, structured parallel programming with deterministic patterns [7,10,11,12,17] is a high-level approach based on a collection of reusable and recurrent execution patterns, also known as skeletons, which offers better abstraction and hides low-level threading details. XPU [1,16] is a C++ framework which falls in this category and try to develop this approach toward better productivity and higher performances. In this paper, we show how we can use XPU to ease parallelism expression in a radar signal processing application, we show how we can use its different execution patterns to express several different types of parallelism at different levels of granularity. We outline the provided programmability and we shows the achieved speedup by the parallel version in comparison with the original sequential version.

II. XPU Overview

XPU is a task-based library which exploits C++ meta-programming capabilities [6,8,9] to ease parallelism expression. C++ Meta-programming techniques exploits the potential of standard C++ language without any extension and thus does not introduce any need to any particular tool other than a standard C++ compiler. Thus, XPU based programs are simply compiled like any C++ program. XPU provides a friendly and light weight programming interface which enable programmer to design parallel applications or parallelize those which are sequential at the cost of a little amount of extra-code. XPU is a based on a set of recurrent parallel execution patterns which specifies execution configuration of a group of tasks. In order to promote reuse of sequential legacy code without almost any alteration, these tasks are designed to encapsulate different pieces of code including functions, class method or lambda expression. The XPU's execution patterns handle transparently many parallel-paradigms-related routines such as synchronization, communication or task scheduling. An intelligent run-time system exploits information, extracted transparently from both hardware system and used execution patterns (task ordering and task-data dependencies) to

perform dynamically efficient execution on the underlying architecture through cache-aware and load-balanced task scheduling. Contrary to many parallel programming models which introduce new languages, extend existing language or define compiler annotations and thus require a specialized compiler, extra-hardware or virtual machines [5,19]... XPU is a pure software technology entirely based on the traditional C++ language and requires nothing more than a standard C++ compiler to be used, and therefore, improves learning curve steepness and is easily portable to many systems.

III. Task-Based Parallel Execution Patterns

XPU provides a set of parallel execution patterns which enable programmer to express several different type of parallel programming including task, data and pipeline parallelism at different granularity level.

A. Task definition

In XPU tasks can be defined from different pieces of C/C++ code including functions, class methods or lambda expression. This task design enable the programmer to reuse sequential legacy code to build a parallel application. Figure 1 shows an example of how we can use a function to define a task at the cost of a single line of code.

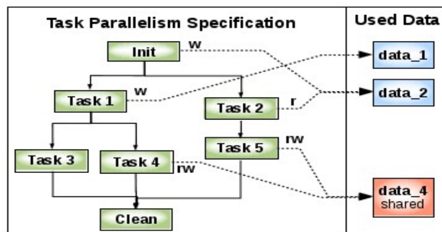
```
1  // original function
2  int filter(float * samples, float size, float cutoff);
3
4  int main() {
5  // task definition
6  xpu::task filter_t(filter, data, size, freq);
7 }
```

Figure 1. A function can be easily reused to define a task without altering the original code

Once defined, a set of tasks can be used within and execution pattern which specify their parallelism.

B. Task parallelism

Parallelism between tasks can be easily specified using the “parallel” keywords, as shown in Figure 2. Since some of these tasks might be sequential due to some consumer-producer dependencies, the “sequential” keyword in conjunction with “parallel”, allows the specification of both parallel and sequential execution in a single task graph.




```
1 void main() {
2  task t1(function, data_1),       // task definition
3       t2(&o, cls::method, data_2), ...;
4  task_group * program; // task graph
5  program = parallel(sequential(t1, parallel(t3,t4)),
6                     sequential(t2, t5));
7  init();
8  program->run(); // 'data_4' protected automatically
9  clean();
10 }
```

Figure 2. An example of task graph composed of parallel and sequential tasks

C. Data Parallelism

1) Parallel loop

Data parallelism refer to a execution configuration where the same task is performed repetitively for each item of a large amount of data. The parallel for loop “parallel_for” is a typical execution pattern which exploit data parallelism to offer a great parallelism multiplier. Figure 3 shows an example of a parallel loop construct.

```
1  int process(int from, int to, int step, float ** pulses) {
2    for (int i=from; i<to; i+=step)  filter(pulses[i]);
3  }
4  void main()
5  {
6    float ** pulses = … ;
7    task process_t(process, 0,0,0, pulses);
8    parallel_for pf(0, pulse_count, 1, &process_t);
10   pf.run();
11 }
```

Figure 3. Parallel for loop construct example

2) Vectorization

Vectorization can act as a great performance multiplier by allowing SIMD (Single Instruction Multiple Data) operation on a vector of homogeneous data. XPU provides transparent vectorization through a built-in vectorized type named “vec4f” and implemented using x86 SSE intrinsics. “vec4f” allows vectorization of standard float operations as well as many other functions.

XPU provides others data parallel execution patterns[1] which are not exposed here since they will not be used on the target application.

D. Temporal parallelism : Pipeline

Pipeline parallelism [13,14,15] is a recurrent execution pattern in many applications implying real-time stream processing such as digital signal processing and video processing [18,20,21]. The Pipeline construct implement a consumer-producer relationship between a set of tasks often referred as “stages”. Parallelism is exploited only of independent activities while serial execution is enforced on dependent ones. Figure 4 shows how a pipeline can be expressed in XPU.

```
1 void sharpen(int i, vector<image> * imgs) // i = frame index
2 { imgs[i]->sharpen(); }
3
4 void multiply(int i, vector<image> * imgs, image * mask)
5 { imgs[i]->multiply(mask); }
6
7 int main()
8 {
9   vector<image> frames(size);
10  ...
11  task sharpen_t(sharpen, 0, &frames),
12       blur_t(blur, 0, &frames),
13       multiply_t(multiply, 0, &frames, &mask);
14 task_group * process_image = pipeline(size, sharpen_t,
15                                            blur_t,
16                                            contrast_t,
17                                            multiply_t);
18  process_image->run(); // frame index “i” will be updated
```

Figure 4. Pipeline parallelism expression in XPU : an example of four stages image processing pipeline

IV. Study Case study : real-time Radar Signal Processing

A. Algorithm overview

The target application is a radar signal processing algorithm which processes a digitized signal of a phased array radar system. Data volume grows significantly as enabled channels count grows. As shown in figure 5, the algorithm perform its task in eight steps. These steps may be summarized in three major steps which are:

1- Digital Beam Forming (Green)
2- Doppler Filtering (Blue)
3- Pulse Compression (Red)

Echos are received as periodic bursts. For each burst, the received signal's samples feeds the signal processing chain which must perform all required operations before the next burst is received in order to meet real-time processing.

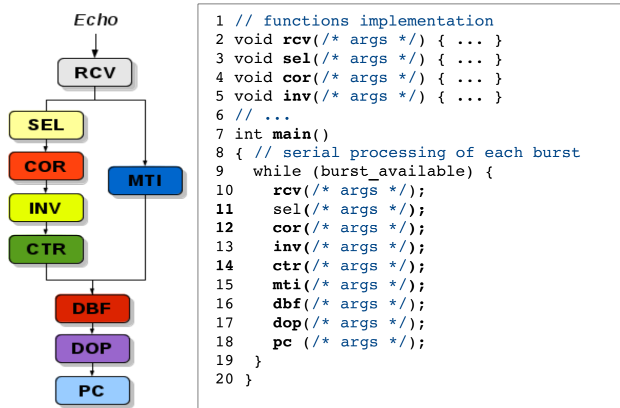


Figure 5. Overview of a radar signal processing algorithm and its sequential implementation (arguments have been omitted for brevity)

The first task “RCV” is responsible of data reception from the radar simulator and introduces thus negligible workload. The four following blocks: SEL (Selection), COR (Correlation), INV (Inversion) and CTR (Control) perform a correlation of the receiver channels and guide digital beam forming. The MTI (Moving Target Indication) block allows the discrimination of moving targets against stationary clutter. DBF (Digital Beam Forming) forms beams using the processed input channels. Doppler processing is then performed using these beams by the DOP (Doppler) block and finally Pulse compression is completed by the last block PC (Pulse Compression).

The target Radar can be configured to use different number of input channels. Input data volume as well as computing load is proportional to the enabled channels count. While Figure 7 shows the theoretical computing load of each processing block for one 64 channels-burst, Figure 6 depicts the generated input/output data volume for/by each processing block by the same configuration.

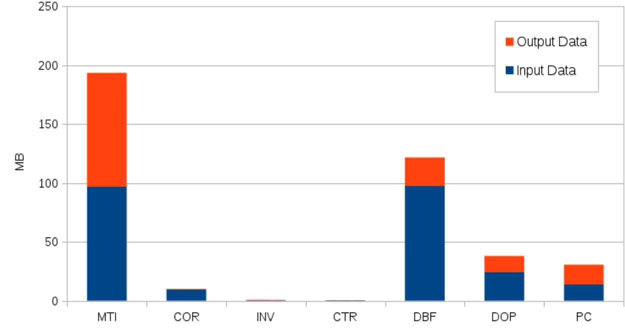


Figure 6. Input/Output Data volume per burst for each processing block in a 64 channels configuration

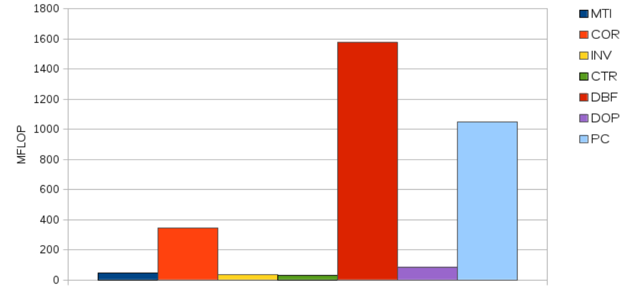


Figure 7. Computing load of each processing block for a 64 channels-burst (Floating-point Operations).

B. Experimental Setup

1) Hardware Setup

In the next sections, we use two different execution platforms with one and two general-purpose multicore processors:

- The first platform is an **Intel Core i7 Q720 1.6 GHz** (Max Turbo Frequency: 2.8 GHz), it contains 4 Physical Cores and 8 Threads and has a 45W Maximum TDP (Thermal Design Power) as specified by the constructor[22].

- The second platform is an SMP platform with two **Intel Xeon E5620 2.4 GHz** (Max Turbo Frequency: 2.66 GHz). Each processor has 4 Physical Cores and 8 Threads. Its constructor declares a maximum TDP of 80W per-processor, i.e., about 160W for our target platform [23].

2) Software Setup

We consider mainly three radar configurations of 64, 32 and 16 channels. Total processing time must be equal or lower than 20 ms in the worst case in order to meet real-time processing requirements without loosing any data. Since “DOP” and “PC” tasks performs many FFT (Fast Fourier Transform) in their processing, we use the free FFTW 3.3 library [24] in the case of the PC task. DOP's FFT is much smaller and is written by us.

Serial FFTW is used in the sequential version of PC task and threaded FFTW is used in the parallel one The GNU compiler v4.6.3 is used to compile the different version of the target application. Finally, applications are executed on a Linux Debian OS with the 3.0 Linux Kernel.

C. *Serial Execution*

The initial basic C++ implementation of our processing chain is fully serial. Figure 8 shows the execution time of the entire serial processing chain for different workloads (16, 32 and 64 Channels-Burst) and depicts the spent execution time by each processing block of our algorithm. As expected, the application's execution-time is dominated by DBF, PC and COR tasks. Parallelization should target particularly these blocks in order to reduce the over-all execution time. This basic sequential implementation takes about 7 seconds to process a 64 channels-burst with a fixed beam count on an SMP platform with two Intel Xeon E5620 at 2.4 GHz. So, it runs 350 times slower than the required real-time processing time (20 ms). We outline that the digital beam forming task “DBF” generates the same beam count disregarding the input channels count. This explains the near-constant execution time of the two last blocks “DOP” and “PC” which have a constant workload in all configurations.

D. *Parallelization Methodology*

In order to parallelize our application we follow a simple parallelization methodology which we detailed in previous work [1]. This parallelization technique pass through three main steps which are:

- Decomposing our application into a set of tasks
- Specifying the parallelism of these tasks.
- If possible, parallelizing each of these tasks: each task may be decomposed into finer grain tasks to express finer grain parallelism.
- Finally, instruction-level parallelism (SIMD) may be expressed using vectorized types or concurrent GPU vectors .

Figure 9 gives an overview of our parallelization methodology. We outline that XPU allows hierarchical expression of several types of parallelism including task parallelism, data and temporal parallelism at different level of granularity at both thread level and instruction level.

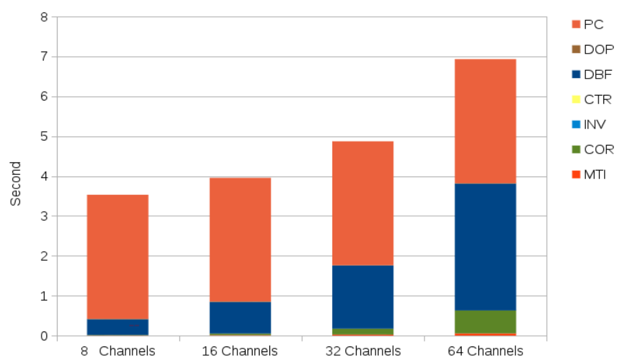


Figure 8. Execution time of the initial serial implementation for 16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz (16 Threads / 8 Cores).

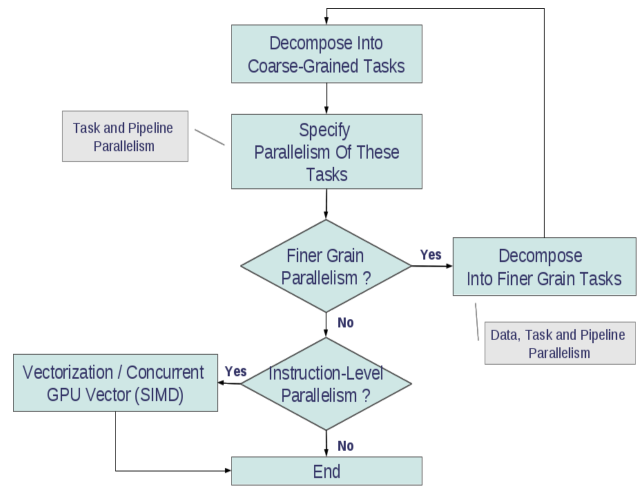


Figure 9. Overview of the XPU Parallelization Methodology

E. *Task Parallelism Extraction*

A first look to the algorithm enables us to distinguish clearly an inherent parallelism between the MTI (Moving Target Indication) and the four following blocks: SEL (Selection), COR (Correlation), INV (Inversion) and CTR (Control). Exposed task parallelism in our study case is relatively weak since the five targeted tasks are not much time-consuming. Consequently, expected speedup is relatively low but nevertheless important. We note that task parallelism may be much more available and effective in other study cases.

Assuming we have a function which represents each of these processing blocks, an XPU task is defined for each of these serial functions through a single line of code. Once task defined, their trivial execution configuration, described in Figure 5, can be expressed at the cost of another single line of code as shown in Figure 10.

Since MTI and SEL accesses to a common input (the burst) respectively in write and read mode, XPU run-time will transform both of them into critical sections in order to protect the burst data from conflictual concurrent accesses (also known as “race condition”) to ensure a safe and coherent execution. This may annihilate parallelism in our case, so, we choose to duplicate input data to preserve parallel execution of MTI and the other four concurrent blocks.

```
1 void main() {
2  // task definition
3  task rcv_t(&rcv, burst_count),
4       mti_t(&mti, args), ...;
5  task_group * burst_processing; // task graph
6  burst_processing =
7     sequential(rcv_t,
8                parallel(sequential(sel_t,cor_t,inv_t,ctr_t),mti_t),
9                dbf, dop, pc);
10 while (burst_available)
11  burst_processing->run(); // parallel processing of each burst
12 }
```

Figure 10. Expression of both parallel and sequential execution of the different tasks using XPU

As expected, parallelization of the sequential processing tasks does not provide a significant speedup since MTI, SEL,

COR, INV and CTR are not time-consuming in comparison with DBF and PC. Nevertheless, this unavoidable parallelization step reduces slightly the over-all execution time as shown in figure 11. The gained execution time grows proportionally to the input data size. In order to make this task-parallelism speedup significant, COR,DBF and PC execution times must be significantly reduced to be as closer as possible to the MTI execution time.

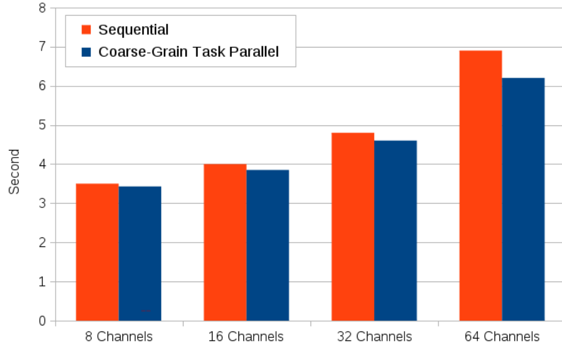


Figure 11. Execution time of both the sequential version and the task-parallel version for 8,16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz

In the next sections, we try to parallelize each block at thread and instruction level through XPU data parallel execution patterns.

F. Data Parallelism

As we stated earlier, this algorithm processes simultaneously all data (signal's samples) received from several channel of the phased array radar system. Many blocks of the algorithm performs the same operation on each channel. This can be exploited to implement data parallelism in each of these stages at instruction-level by using the vectorization capabilities of XPU and at thread-level through replacing sequential loops by parallel ones using the "parallel_for" execution pattern available in XPU. COR, DBF and PC should be particularly targeted by this parallelization.

1) Vectorization

Vectorization can be implemented by replacing regular simple precision float by the "xpu::vec4f" built-in type which translates transparently simple operations on floating-point into SIMD operations on four floats simultaneously. This instruction-level parallelism is implemented on top of x86 SSE streaming intrinsics and may be extended in future works to support other SIMD extensions such as AVX or Altivec. We note that data should be aligned in memory to make vectorization efficient. This can be performed through the XPU's aligned allocator which is compatible with standard C++ STL containers.

Vectorization has been used in MTI, COR, DBF and PC tasks. Figure 12 shows that instruction-level parallelism can be a great parallelism multiplier for data parallel tasks such as MTI and COR where the same operation is performed on large vectors of samples. We note that SIMD impact vary depending on the workload size implying both data size and computing load: the vectorized COR code performs 10 times faster than the original sequential code while the vectorized MTI code achieve only about 20% execution speedup.

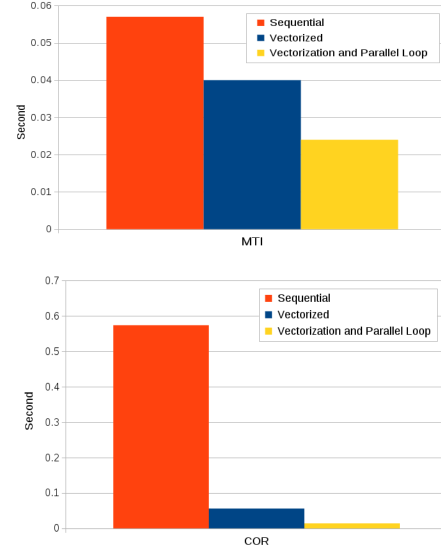


Figure 12. Thread-level and Instruction-level parallelization effect on execution-times of MTI and COR for 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz

2) Parallel loop

Instruction-level parallelism offers limited scalability since SIMD may take advantage of processor frequency increasing but cannot benefit from processor count increasing without thread-level parallelism. Parallel loop implements data parallelism at thread-level and provides good scalability on multicore architectures. We use XPU's "parallel_for" execution pattern in conjunction with vectorization to parallelize several tasks. The XPU's "parallel_for" construct adapts dynamically to the underlying architecture to exploit all available processors.

Figure 13 shows an example of parallel for implementation to perform simultaneously several FFT on several pulses (pulse: a set of samples). The "parallel_for" construct has been used to parallelize MTI, COR, DBF, DOP and PC tasks. The vectorized code has been reused in the parallel loops. Figure 12 depicts a significant speedup in comparison with the initial serial version: the parallelized COR version runs about 48 times faster than the original sequential version and 5 times faster than the vectorized version on the bi-processor 16 Threads SMP platform.

```
1  int fft(int from, int to, int step, float ** pulses)
2  {
3    for (int i=from; i<to; i+=step)
4                   cplx_fft(pulses[i]);
5  }
6
7  void pc(float ** pulses, int pulse_count)
8  {
9    task fft_t(fft, 0,0,0, pulses);
10    parallel_for p(0, pulse_count, 1, &fft_t);
11   p.run();
12 }
```

Figure 13. A simplified example of parallel for loop use in the PC task.

3) *Performances*

As shown in Figure 14, after parallelizing the most time-consuming blocks of our processing chain at thread and instruction-level, the over-all processing time of a 64 channels-burst on the dual Intel Xeon E5620 platform dropped from 6.9 seconds (initial sequential version) to 0.045 second (in the worst case) for the parallel version as depicted Figure 15. The 64 channels configuration does not satisfies the real-time requirement (20 ms in the worst case). However, we are able to process input data in real-time in the 32,16 and 8 channels configurations on the same platform.

With a maximum TDP of 160 Watts, the first platform may not be embeddable due to potential energetic constraints. Thus, a platform based on one Core i7 Q720 with a maximum TDP of 45 Watts may be more suitable. We executed our application on this platform. Figure 15 shows that our application run twice as slow as on the SMP platform. Consequently the 32 channels configuration does not allow real-time processing on that platform. However, real-time processing can be performed in the 8 and 16 channels configurations.

As shown in figure 14 and 15, the current application displays a good scalability. We believe that real-time processing in all our four configurations may be achieved on faster platforms with more processing cores and higher clock frequency.

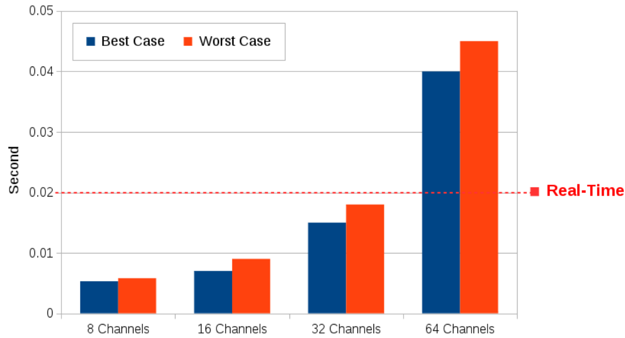


Figure 14. Execution time achieved by the parallelized version for 8,16,32 and 64 channels-burst on 2 x Intel Xeon E5620 2.4 GHz

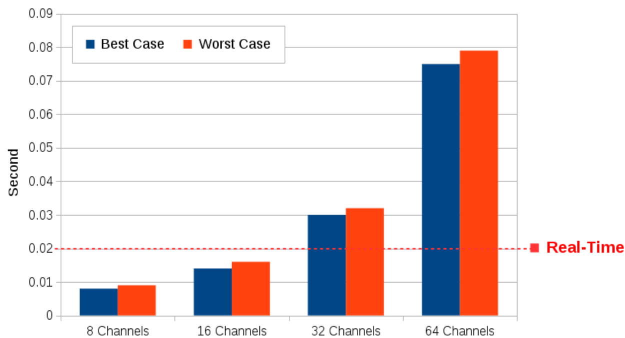


Figure 15. Execution time achieved by the parallelized version for 8,16,32 and 64 Channels-burst on an Intel Core i7 Q720 1.6 GHz

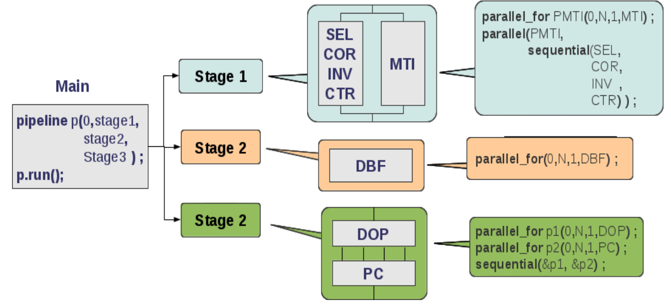


Figure 16. Overview of the final program architecture and code (details of the code have been omitted for clarity).

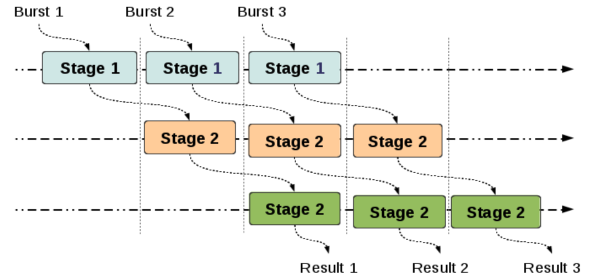


Figure 17. Execution configuration of three-stages-pipeline processing three consecutive bursts

As we have seen, data parallelism can offer significant execution speedup especially when coupled with task parallelism. However, available parallelism may be limited by producer-consumer dependencies between tasks. In this case, temporal parallelism (a.k.a. pipeline parallelism) can be a very useful parallelism multiplier.

G. *Pipeline Parallelism*

At the opposite or serial execution pattern where all tasks are executed sequentially, pipeline exploits the available parallelism by executing simultaneously all its processing stages and serializing only the dependent activities. This allows potential throughput improvement especially in the case of applications involving real-time continuous stream processing such as in our case.

In our application, a single burst must be processed by the different tasks in the specified order to preserve data coherency: these tasks are acting as a consumer-producer chain. However, multiple independent bursts can be processed simultaneously by different processing stages without violating the producer-consumer dependencies as shown in Figure 17.

We use the “pipeline” execution pattern of XPU to implement a pipeline execution configuration. As illustrated in Figure 16, we regroup our tasks into three processing stages:

- **Stage 1** holds MTI, SEL, COR, INV and CTR tasks. We note that the “parallel” construct specifies

parallelism between MTI and the other four tasks which are executed sequentially. In addition, we use two “parallel_for” loops in MTI and COR in conjunction with vectorization.

- **Stage 2** contains the parallel version of DBF : a “parallel_for” loop enables parallel processing of different channels.
- **Stage 3** is composed of DOP and PC tasks. Both of them uses “parallel_for” loops to process simultaneously several beams.

This task regrouping pattern aims to load-balance the pipeline stages. A FIFO (First In First Out) queue ensure data transfer between different stages. The FIFO sizes are limited by the available memory and may have a non-negligible impact on the achieved performances.

Figure 17 illustrates how the three stages of the pipeline can process simultaneously three different bursts without violating producer-consumer dependencies. The different stages may be executed differently depending on the workload of each stage however stage’s ordering is preserved for each received burst.

As shown in Figure 18, pipeline parallelism allows real-time processing in all our four configurations on the 8 threads platform (Intel Core i7 Q720). By observing the load of the different processor's cores when executing the application as well as the achieved execution times, we can conclude that the pipeline-based version exploits the computing resources more efficiently than the previous parallel version.

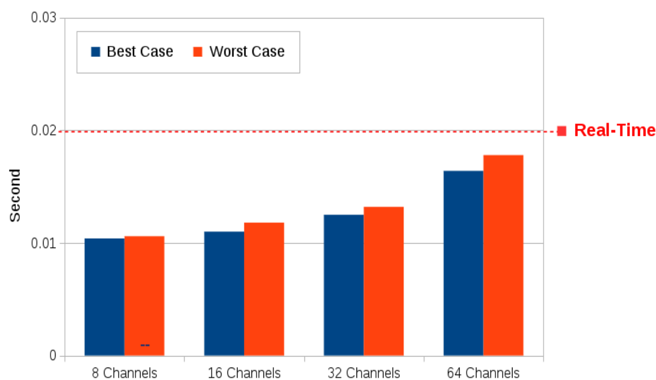


Figure 18. Worst and best execution time of the parallel version with pipeline parallelism version execution for 8,16,32 and 64 Channels-Burst on the Intel Core i7 Q720 1.6 GHz (8 Threads)

V. Conclusion

As we have seen previously, we parallelized our target application progressively by expressing several parallelism types at different granularity levels starting from coarse grain tasks to finer grain ones. Use of both task, temporal and data parallelism allowed us to extract a significant amount of parallelism and to achieve high performance and good scalability. We outline the programmability of the XPU framework which enables programmer to easily express several types of parallelism at all levels of granularity at the cost of a little amount of parallelism related extra-code and in the same time, by enabling him to reuse most of the legacy sequential code without significant alteration. This programmability can improve significantly programmer's productivity.

By easing parallelism expression on general purpose multicore architectures such as the x86 architecture, many real-time digital signal processing applications are likely to be implemented efficiently on such architectures making them an attractive alternative to expensive specialized processing architectures such as FPGA and DSP based platforms.

References

[1] N. Khammassi, J.C. Le Lann, J.P. Diguet and A. Skrzyniarz, “MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology”, HPCC '12 Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication, 71-80, Liverpool UK, June 2012

[2] G. Blake, R. G. Dreslinski and T. Mudge, “A Survey of Multicore Processors”, IEEE Signal Processing , vol. 26, n. 6, pp. 26-37 November 2009

[3] L. J. Karam, I. Alkamal, Alan Gatherer, G. A. Frantz, D. V. Anderson and B. L. Evans, “Trends in Multicore DSP platforms”, IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[4] W. Wolf, “Multiprocessor System-on-Chip Technology”, IEEE Signal Processing vol. 26, n. 6, November 2009

[5] H.Park, H. Oh and S. Ha “Multiprocessor SoC Design Methods and Tools”, IEEE Signal Processing vol. 26, n. 6, November 2009

[6] H. Singh, “Introspective C++”, Thesis, Virginia Polytechnic Institute, 2004H. Singh, “Introspective C++”, Thesis, Virginia Polytechnic Institute, 2004

[7] M. D. McCool, "Structured Parallel Programming with Deterministic Patterns", HotPar'10 Proceedings of the 2nd USENIX conference on Hot topics in parallelism, 2010

[8] IJ. Koskinen, “Meta-programming in C++”, March 9, 2004

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In CC ’02. Springer- Verlag.

[10] M. Aldinucci and M. Danelutto, “Skeleton-based parallel programming: Functional and parallel semantics in a single shot”, Comput. Lang. Syst. Struct., 33(3-4). 2007, pp. 179-192

[11] M. Cole, “ Algorithmic Skeletons: structured management of parallel computations”, Pitman/MIT Press, 1989

[12] M. Cole, “Bringing Skeleton out of the closet: a pragmatic manifesto for skeletal parallel programming”, Parallel Computing , 30(3), pp. 389-406, March 2004

[13] E. C. Reed, N. Chen and R. E. Johnson, “Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't”, Proceeding SPLASH '11 Workshops Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11 Pages 133-138, 2001

[14] S. Macdonald, D. Szafron and J. Schaeffer,“Rethinking the pipeline as object–oriented states with transformations”, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS’2004) at IPDPS, 2004

[15] C. Deloatch, “Pipeline Parallelsim”

[16] XPU Framework, “http://www.xpu-project.net/”

[17] Horacio González-Vélez and Mario Leyton "A survey of algorithmic skeleton frameworks: high-level structured parallel programming

enablers" Software: Practice and Experience Volume 40, Issue 12, pages 1135-1160, November/December 2010.

[18] D. Lin, X. Huang, Q. Nguyen, J. Blackburn, C. Rodrigues, T. Huang, M. N. Do, S. J. Patel and W. W. Hwu, “The Parallelization Of Video Processing”, IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[19] Hahn Kim and Robert Bond, “Multicore Software Technologies”, IEEE Signal Processing, vol. 26, no. 6, pp. 80-89

[20] S. T. Klein and Y. Wiseman, “Parallel Huffman Decoding”, Proceeding DCC '00 Proceedings of the Conference on Data Compression IEEE Computer Society Washington, DC, USA 2000

[21] J. Kepner and J. Lebak, “Software technologies for high-performance parallel signal processing”, Lincoln Lab. J., vol. 14, no. 2, pp. 181-198, 2003

[22] “Intel Core i7-720QM”, http://ark.intel.com/products/43122/

[23] “Intel Xeon E5620”, http://ark.intel.com/products/47925/

[24] “FFTW Library”, http://www.fftw.org/