Parallelizing Ultrasound Image Processing using OpenMP on Multicore Embedded Systems

Lei Huang
Computer Science Department
Prairie View A&M University
Prairie View, Texas 77446
lhuang@pvamu.edu

Eric Stotzer
Texas Instrument
Houston, Texas
estotzer@ti.com

Hangjun Yi, Barbara Chapman
and Sunita Chandrasekaran
University of Houston
Houston, Texas
hangjun05@gmail.com, bchapma3@mail.uh.edu,
sunita@cs.uh.edu

***Abstract*—The shift towards multicore architectures poses significant challenges to the programmers. Unlike programming on single core architectures, multicore architectures require the programmer to decide on how the work needs to be distributed across multiple processors. In this contribution, we analyze the needs of a high-level programming model to program multicore architectures. We use OpenMP as the high-level programming model to increase programmer productivity, reduce time to market and development/design costs for these systems. In this work, we have explored the medical ultrasound application using OpenMP on a TI-based Tomahawk platform that is a six-core, high performance multicore DSP system. This application is heavily based on image processing and the goal is to achieve desired level of image quality. We have explored the different cache configurations of the system. In this process, we were able to study the performance impacts of data locality when data objects are placed into different components of the Tomahawk memory system.**

I. Introduction to Embedded systems

Embedded processors are being extensively used in several applications belonging to different domains that include telecommunication systems, robotics, automotive systems, medical applications and so on. Multicore processors consist of many cores on a single chip, this is primarily to overcome resource constraints of single processor system and also be able to deliver better performance. Although hardware technology seems to be improving, there are not sufficient software toolsets that could exploit the parallelism offered by the hardware. This leads to a potential gap between hardware and software engineers; products tend to get more complex and software toolsets are unable to cope.

Important features of a multicore processor are the interconnect and communication model. The interconnect is the physical hardware that is used to transport data between system resources. The type of interconnect determines the communication model, which is either shared memory or message passing. Distributed systems have processors with private memories and an interconnect that passes data as messages or packets. Likewise, shared memory systems have memories that are accessible by all processors. Data is passed between processors via the shared address space. Shared memory systems must implement some type of cache coherency, a memory consistency model, and synchronization primitives. The parallel programming model defines the application programmers interface (API) for expressing application parallelism. Different parallel programming models are appropriate for different types of interconnect and communication models. Shared-memory models must implement a shared address space, provide a method to start and control threads and control access to shared data.

There are two classes of multi-processing (MP) systems: Symmetric and Asymmetric multi-processing (SMP and AMP). The former has multiple processors or cores sharing a common view of main system memory, processors are generally identical which enables the programmer to assign threads to different processors depending on the load balancing conditions. The advantages of the common memory model can be exploited only if all the dependencies are met without deadlock or starvation and the processing is done in the right order. The AMP model has much more loosely coupled, quite different Instruction Set Architectures (ISAs) and dedicated local memory resources. Here the processors are tuned to specific tasks and the same requirements such as meeting dependencies and preserving correct ordering of computations like that for SMP holds good.

Parallelism seems to be no longer restricted to homogeneous models, but it appears to be spreading along different system levels that includes nodes, cores, threads and other vector units. MPSoCs have emerged as an important classification of VLSI system, these are not simple traditional processors but allow heterogeneous configurations. MPSoCs are designed to fulfill the requirements of embedded applications. The most commonly used languages to program embedded systems are assembly level languages and C languages. Assembly languages are difficult to use and they are machine-dependent. In embedded market, compilers are not 100% ANSI C compliant primarily due to two reasons. A. Some of the C features are complicated to implement on the embedded processor. B. In some cases vendors may require to extend C language because of the need to support special features on the chip.

Programmers find these C-based languages difficult to learn, hence it is a challenge to exploit the underlying platform. Moreover a code written once using a particular C-extended language cannot be used on more than one type of processor, even if the processor is from the same family. Other language

978-1-4673-5085-3/12/$31.00 ©2012 IEEE

extension-based approaches try to abstract the low-level details of the platform from the programmer. We see that there is a real challenge to provide the right kind of software for an embedded platform.

In this paper we discuss an approach to design and construct an effective programming model for embedded systems in order to reduce the design cost, reduce development time, facilitate experimentation with a variety of design choices and promote software reuse. We will discuss in detail the implementation challenges of a high-level programming interface for MPSoCs in the forthcoming sections. The goals of this research are:

- To identify the challenges in adapting OpenMP to distributed memory architecture
- To explore ultrasound medical imaging algorithm and parallelize the same using a multicore high performance DSP system.
- To explore the memory configuration of the target system

A. Emerging standards to program embedded systems

Heterogeneous multicore embedded systems are often comprised of special purpose devices that have their own functionalities and instruction sets along with the general-purpose cores. A set of tools and programming models is necessary in order to efficiently use the computational capabilities of these multicore embedded systems. A uniform and general-purpose programming model that enables the user to manage the low-level interfaces of different devices, mostly embedded, and that facilitates application programming on several types of embedded devices, would be ideal.

In 2005 the Multicore Association (MCA) [1], [2], [3] was formed by a group of vendors and software companies to address the challenges persistent in the heterogeneous multicore environment. MCA is providing standards/APIs for inter-process communication (MCAPI) and resource management (MRAPI). The association also has an active working group to provide a tasking API (MTAPI). University of Houston has been active members in the working group contributing towards the design of MTAPI specification. Although intended to facilitate portability, these are low-level APIs that could be tedious for application programmers. As a result, it would be worthwhile to consider a high-level programming model such as OpenMP [4] that can abstract low-level complexities of heterogeneous systems from the programmers. Extensions are being considered to extend OpenMP for accelerators by the OpenMP ARB. Another effort, OpenACC provides similar solutions and targets accelerators (GPUs)[5].

II. OpenMP

OpenMP is a widely-adopted shared memory parallel programming interface providing high level programming constructs that enable the user to easily expose an applications task and loop level parallelism in an incremental fashion. Its range of applicability was significantly extended by the addition of explicit tasking features [6]. OpenMP has already been proposed as the starting point for a programming model for heterogeneous systems by Intel [7], ClearSpeed [8], The Portland Group [9] and CAPS SA [10]. The idea behind OpenMP is that the user specifies the parallelization strategy for a program at a high level by annotating the program code; the implementation must then work out the detailed mapping of the computation it contains to the machine. It is the users responsibility to perform any code modifications needed prior to the insertion of OpenMP constructs. In particular, it requires that dependencies that might inhibit parallelization are detected and where possible, removed from the code. The major features are that the directives that specify a well-structured region of code should be executed by a team of threads, who share the work. Such regions may be nested. In order to effect a distribution of work among the participating threads, worksharing directives are provided. In a related work [11] we have extended this standard with the notion of a subteam, which permits work to be shared among a subset of the threads that are executing a parallel region. Additional constructs and library functions provide a finer degree of control over the execution of the program if needed.

A. Implementation

We have seen that OpenMP is one of the two major models for parallel programming, the other one being message passing interface (MPI) model. There are several different OpenMP implementations or compilers that provide OpenMP. [12] describes the effective OpenMP implementation and translation for MPSoC without using OS. The target platform has physically shared memories, hardware semaphores and no OS. Their effort concentrates on translation of global *shared* variables, implementation of the *reduction* clause and synchronization directives. Task based implementations in the nano-threads [13] programming model represent program as task graphs representing hierarchical parallelism. As part of our earlier work [14] we used SMARTS runtime system to create data flow execution model using a task based translation. Our own work has shown the usefulness of compile-time analyses to improve data locality (including performing a wide-area data privatization [15]) as well as to reduce synchronization overheads statically. An OpenMP-aware parallel data flow analysis would enable the application of a variety of standard (sequential) optimizations to an OpenMP code [16]. IBM provides an OpenMP compiler [17] that presents the user with a shared memory abstraction of the Cell B.E. [18]. Their work focuses on optimizing DMA transfers between the SPEs local stores. This abstraction results in reduced performance, but creates a more user-friendly abstraction. GOMP [19] is the OpenMP implementation for GCC, that implements its runtime library (libGOMP) as a wrapper around the POSIX threads library, with some target-specific optimizations for systems that support light-weight implementation of certain primitives.

B. OpenMP for embedded architectures

Implementation of OpenMP on MPSoC is discussed in [20]. MPSoC consists of multiple RISC and DSP cores. This work proposes language extensions to explicitly specify whether a

parallel region will execute on a RISC or DSP platform. DMA transfers are supported directly with language extensions. [21] developed a cross compiling environment for OpenMP implementation to target multicore processors such as M32700, MPcore, RP1 for embedded systems and Core2Quad Q6600 for a desktop PC. It is inferred that multi-core processors for embedded systems have larger synchronization cost and slower memory performances than for desktop PC. They use spinlock mechanism to improve synchronization performance. However it is important to remember that if the spinlock mechanism is not carefully designed, it can easily lead to deadlock situations. Also a spinlock for longer duration can reduce the system performance. The authors discuss parallelization efforts using OpenMP-based high-level language on multiprocessor platforms like ARM MPCore in [22]. The authors exploit capabilities of signal processing algorithms and parallel computation capabilities.

C. OpenUH

The compiler that we use to develop our current work is based upon the OpenUH compiler, a branch of the open source Open64 compiler suite for C, C++, and Fortran 95 developed at the University of Houston. OpenUH contains a variety of state-of-the-art analyses and transformations, sometimes at multiple levels. Its interprocedural array region analysis uses the linear constraint-based technique proposed by Triolet [23] to create a DEF/USE region for each array access; these are merged and summarized at the statement level, at basic block level, and for an entire procedure. Dependence analysis uses the array regions and procedure summarization to eliminate false or assumed dependencies in the loop nest dependence graph. Both array region and dependence analysis use the symbolic analyzer, based on the Omega integer test [24]. We have enhanced the original interprocedural analysis module [14], designing and implementing a new call graph algorithm that provides exact call chain information, and have created a GUI called Dragon [25] that enables an application developer to request and view information on a submitted program and its data structures, typically in the form of graphs or text along with the corresponding source code.

We are currently supporting major portions of OpenMP 3.0 and improving our tasking implementation, along with an explicit cost model for OpenMP in the compiler. The runtime library is based on POSIX threads (PThreads) [26]. OpenUH provides native code generation for IA-32, IA-64 and Opteron architectures. It has a source-to-source translator that translates OpenMP code into optimized portable C code with calls to the runtime library, which enables OpenMP programs to run on other platforms. Its portable, multithreading runtime library includes a built-in performance monitoring capability [27]. Moreover, OpenUH has been coupled with external performance tools to support the analysis and visualization of a programs performance [28]. The compiler is directly and freely available via the web (http://www.cs.uh.edu/openuh).

III. Adapting OpenMP for Embedded Systems

OpenMP is a widely-adopted shared memory parallel programming interface providing high level programming constructs that enable the user to easily expose an applications task and loop level parallelism in an incremental fashion. Its range of applicability was significantly extended by the addition of explicit tasking features [6]. OpenMP has already been proposed as the starting point for a programming model for heterogeneous systems by Intel [7], ClearSpeed [8], The Portland Group [9] and CAPS SA [10]. The idea behind OpenMP is that the user specifies the parallelization strategy for a program at a high level by annotating the program code; the implementation must then work out the detailed mapping of the computation it contains to the machine. It is the users responsibility to perform any code modifications needed prior to the insertion of OpenMP constructs. In particular, it requires that dependencies that might inhibit parallelization are detected and where possible, removed from the code. The major features are that the directives that specify a well-structured region of code should be executed by a team of threads, who share the work. Such regions may be nested. In order to effect a distribution of work among the participating threads, worksharing directives are provided. In a related work [11] we have extended this standard with the notion of a subteam, which permits work to be shared among a subset of the threads that are executing a parallel region. Additional constructs and library functions provide a finer degree of control over the execution of the program if needed.

In our earlier work, we have implemented OpenMP [29] on the OpenUH compiler [30], which is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95 developed at the University of Houston. OpenUH contains a variety of state-of-the-art analyses and transformations, sometimes at multiple levels. In our implementation, we built the OpenMP runtime on top of a light-weight scalable real-time operating system (RTOS), DSP/BIOS developed by TI, which is designed to require minimal memory and CPU cycles [31]. DSP/BIOS implements a priority-based preemptive multi-threading model for a single processor. High priority threads preempt lower priority threads. DSP/BIOS provides for various types of interaction between threads, including blocking, communication, and synchronization. To translate OpenMP code for MPSoC model, we use OpenUH compiler and perform source-to-source lowering of the OpenMP constructs into explicit multi-threaded C code, including calls to its OpenMP runtime library. The lowered C source files that are produced by OpenUH have a `.w2c.c` extension. The lowered files are then compiled with the TI C6X compiler. Using the TI C6X linker, the resulting object files are linked with a modified OpenMP runtime library, with modifications to use DSP/BIOS and manage cache coherence. More detailed implementation can be found in paper [29].

IV. Medical Applications on Embedded Systems

Medical imaging is a set of techniques that involve creating images of the human body for diagnosing diseases inferring

the cause from effect (signals), a reverse mathematical problem. Medical ultrasonography and computed tomography are two of the most important medical imaging techniques. Improving the quality of the medical images involves exposure to harmful radiation from X-rays and other imaging techniques, these can prove to be very hazardous to the patient. It remains a challenge to build an efficient medical imaging technique that can construct good quality images and at the same be less hazardous to the patient. Using Multicore embedded processors could be one of the several solutions to provide sophisticated imaging techniques. These processors include multiple special purpose processors such as digital signal processors (DSPs). We see that there might be the right kind of hardware technology available but are there adequate software toolsets to exploit the advantages of these hardware devices?

A. Ultrasound Systems

Ultrasound systems, which employ sound waves with frequency greater than the upper limit of human hearing (20 kHz), have widespread usage in medical diagnosis, including cardiology, obstetrics, gynecology, and abdominal imaging. These sonography is generally described as a safe test because it does not use mutagenic ionizing radiation, overuse of which may cause health hazards. Such devices can take the form of hand held scanners that can be used outside of a doctor's office, to name a few: 3-dimensional real-time imaging systems, and tissue characterization via signal backscatter processing.

An ultrasound machine or the board may be specialized for a specific application. To achieve desired levels of performance and image quality, ultrasonic imaging devices must be equipped with substantial processing capability and software that can fully exploit it. There exists a high degree of parallelism in the critical functions of an ultrasonic imaging system such as signal generation, acquisition, filtering, and image reconstruction. Unfortunately, lack of supporting programming paradigm may lead to redesigning of hardware, which may prove to be costly or replacement of boards and so on. This process is time consuming especially when time-to-market solutions are critical in medical domain. In an extreme case, old boards must be discarded and new boards accommodating new features will be designed, but this is not cost-effective. Health care organizations expect the systems to last longer and would prefer to upgrade the system as and when necessary. As a result, there is an urgent need for a software infrastructure that should provide low-power, low-cost and high performance solution. An overview of the ultrasound system is shown in Figure 1. In the system, the front end receives signals from the scanner, and performance the beamforming algorithm to generate merged data from multiple channels of input. Additional image processing and optimizations including B mode processing, spectral Doppler processing, noise removal and scan conversion are performed at the middle end to generate a stream of images. A detailed explanation of the block diagram can be found in [32].

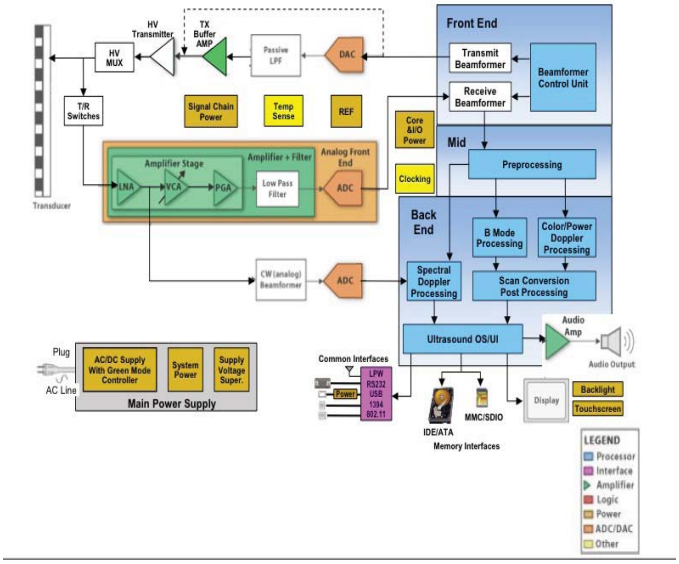

Fig. 1. Overview of Ultrasound system

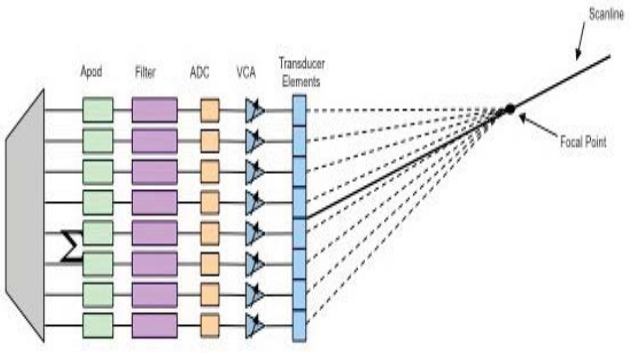

Fig. 2. Ultrasound beamforming

B. Rx Beamformer

In this work we focus on parallelizing the Rx beamforming algorithm which is the first operation conducted in the front end, and is the most computation intensive part of the ultrasound system. As shown in Figure 2, due to the multiple channels from scanner, the objective of the Rx beamformer is to calculate the offsets and focus the received multiple channels of sound wave echos from objects that lie along a given scan line. A common implementation is to delay and sum the received signals from the transducer elements to produce effective focal points along a given scan line. The complexity of the problem comes from the fact that delay values are not integer multiples of the ADC sampling rate. In our algorithm, this problem is solved by using interpolation filters.

In our experiment, we used OpenMP, a high-level programming model, to parallelize the algorithm on top of TI C6472 (Tomahawk) embedded processors. With the advent of low power digital signal processors (DSPs), it is beneficial to em-

978-1-4673-5085-3/12/$31.00 ©2012 IEEE

ploy multiple DSP units in a system to perform intensive computation in parallel. However, the questions remain on such systems are 1) how to write a parallel program productively; 2) how to achieve satisfied speedup. In this paper, we evaluate the beamforming program using OpenMP to answer the above two questions. Our experiments are performed using the OpenUH compiler with embedded OpenMP implementation.

C. Beamforming Algorithm Parallelization and Optimizations

The main components of beamforming algorithm are two continuous nested loops as shown in the code snippet of Figure 3. The first nested loop performs the filtering function for all signals in every channel, and the second loop combines all the filtered data from all channels. The first loop accesses data by samples in the row-major order. The second loop accesses data by channels in the column major.

The first scenario of our experiments is a straightforward implementation without performing any loop level optimizations. We added the *omp parallel for* directive on each nested loop to parallelize the two loops. Note that there is an implicit barrier at the end of each parallel for loop. The *omp parallel for* directive distributes the loop iterations to all threads evenly so that each thread processes partial of data. Figure 3 shows the data partition on four threads. Each thread in the first loop processes the input data in a row-major order; and the thread in second loop merges in the processed data in the column-major order. During the merging process of the second loop, the process requires all samples from all channels (128 channels in our experiment) in order to merge them. All of them are stored in a temporary buffer.

After studied the access patterns, we have tested several optimization techniques to make better data reuse efficiently via loop blocking, loop interchange, and loop fusion. The second scenario of our experiments merged the two continuous loops by performing loop interchange followed by loop fusion. As shown in Figure 4, the input data was accessed and merged in one nested loop. We reduced the parallel for loop to one in this scenario. The optimization led to synchronization issues due to fetching of data among several threads in an irregular way. Since the combining function needs all data from all channels, there is a critical section in the function to perform the merging operation in a sequential fashion to avoid data race. As you can see, all of data accesses in this case are in the row-major order, while these data are stored in column fashion. In this case, although we merged two loops into one, we realized that the two additional overheads are introduced: one is the synchronization, and the other is the bad data access pattern. The additional overheads degraded the performance improvement from loop optimizations.

The third scenario of our experiments achieved the best performance by merging the two loops in column-major order. As shown in Figure5, all threads access the data in column-major, and the merging processing can be done without any synchronizations. This optimization eliminated overheads introduced in scenario II, while we reduced two loops into one, and each thread was able to reuse sample data.

```
1  #pragma omp parallel for
2  for (i=0; i<Nchannels; i++)
3    for (j=0; j<Nsamples; j++)
4      FilterData(i, j);
5  #pragma omp parallel for
6  for (j=0; j<Nsamples; j++)
7    for (i=0; i<Nchannels; i++)
8      CombineData(j, i);
```

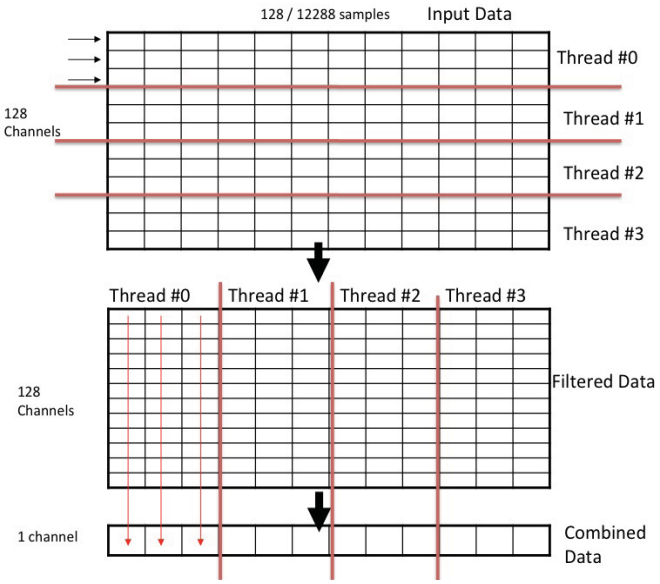

Fig. 3. Scenario #1: Row-major data access in filtering, while column-major data access in combining

```
1  #pragma omp parallel for
2  for (i=0; i<Nchannels; i++)
3    for (j=0; j<Nsamples; j++) {
4      FilterData(i, j);
5      CombineData(j, i); // needs critical section
6    }
```

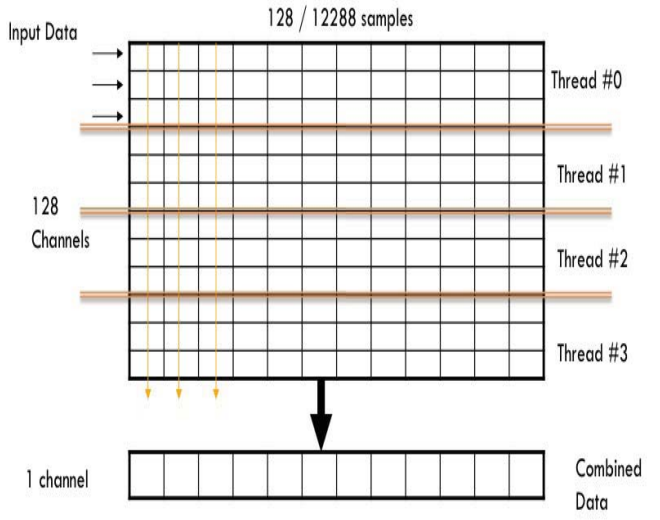

Fig. 4. Scenario #2: Row-major data accessed and combined

```
1   #pragma omp parallel for
2   for (j=0; j<Nsamples; j++) {
3     for (i=0; i<Nchannels; i++)
4     // threads reuse the sample data
5       FilterData(i, j);
6       CombineData(j, i);
7     }
```

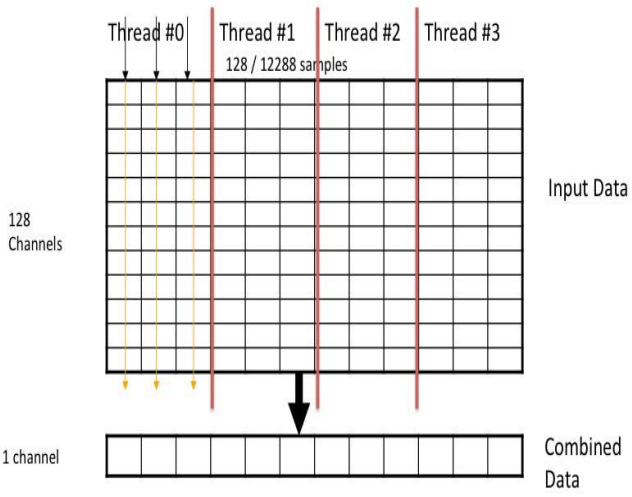

Fig. 5. Scenario #3: Column major data accessed and combined

Threads	Execution time DDR (ms)	Speedup
1	67.449236	1
2	37.094348	1.818316
4	23.921352	2.819625
6	18.414024	3.662928

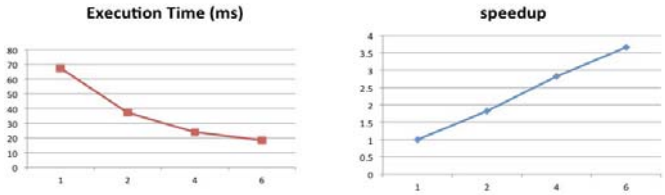

Fig. 6. Scenario #1: Performance on Tomahawk (128x128)channels

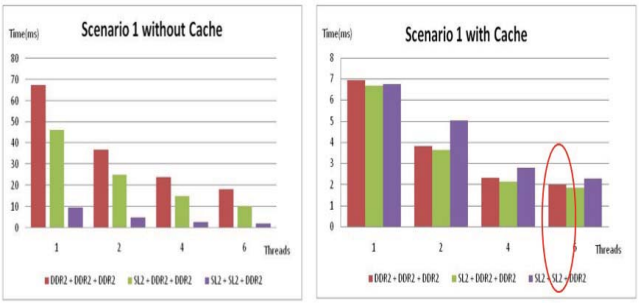

Fig. 7. Scenario #1: Performance on Tomahawk (128x128) channels for different memory configurations

Moreover, we also performed a number of experiments to evaluate the performance impacts of using different memory configurations. The memory configurations include turning the cache on/off, explicitly place data in shared L2 on-chip memory and DDR. The next section gives the details of performance results of the above three cases of experiments.

V. Experiment Results

All of our experiments are executed on a TI-based Tomahawk platform with a six DSP cores. Our first step was to get the parallelized beamformer algorithm working on a multicore embedded system with cache disabled to ensure that our parallelized version was correct. The reason why we disabled the cache first is to avoid any false sharing and cache coherence issues in embedded systems. In this case, all data are placed in DDR memory. We then ran additional experiments with different cache configurations. The L1 cache is private to each core, and we compared the performance of these experiments with L1 cache enabled and with it disabled. We set the L2 cache to shared L2 on-chip memory so that we can control what data are saved in the L2 on-chip memory. By placing different data objects into different components of the C6472 memory hierarchy, we studied the performance impacts of the data locality. The data are stored explicitly configured in the memory hierarchy as the following three types of experiments.

1) DDR+DDR+DDR: All data in the program are placed in the DDR memory as our first experiment.
2) SL2+DDR+DDR: Temporary buffers are placed in the shared L2 on-chip memory, while the rest of data are stored in the DDR memory.
3) SL2+SL2+DDR: Temporary buffers and constant data are placed in the shared L2 on-chip memory, while the rest of data are saved in the DDR memory.

Figure 6 shows the speedup and the execution time of the first type of experiment on the six cores of the TI platform. In this experiment, we placed all data in the DDR memory without using L2 shared memory. Although the overall execution time is very long, we can see that the execution time is significantly reduced while using all the six cores. The first results were quite encouraging in term of scalability.

Figure 7 shows the performance comparison of scenario #1 experiments with/without L1 cache. When the L1 cache is disabled (the left figure), the execution time was dramatically improved after we placed more data into the shared L2 on-chip memory. When the L1 cache is enabled (the right figure), there are almost 10 times difference in execution time, which indicates that L1 cache has improved the data access time dramatically. However, we do not see the significant performance difference with explicit data placement on L2 on-chip memory.

We do not include the performance data of scenario #2, since the performance was very poor, because the memory

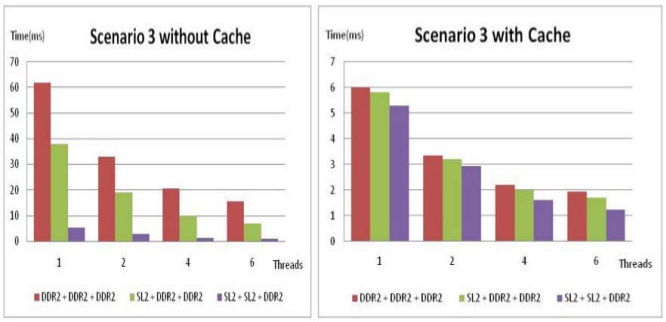

Fig. 8. Scenario #3: Performance on Tomahawk (128x128)channels for different memory configurations

access pattern is orthogonal to the data storage. Figure 8 shows the execution time and speedup of scenario #3 using the three types of data placements. As similar to scenario #1, the performance is very scalable when L1 cache is disabled. However, we achieved additional speed up when L1 cache is enabled, in which we did not observe in scenario #1. The difference comes from the removal of synchronizations, and better data reuse. These figures show that the performance is best obtained when the frequently accessed data are stored in the shared L2 memory as much as possible, and with L1 cache enabled.

VI. Conclusions

In this work, we have discussed the programming challenges on a MPSoC architecture from TI that consists of six TMS320C64x processor cores. We have considered a medical imaging application to be ported on this high performance multicore DSP system. We use OpenMP, a high-level programming model to parallelize the application on the given multicore embedded platform. We also explored the different memory configurations available on this DSP system. In this process, we identified the challenges in adapting OpenMP for a embedded multicore system. We explored different parallelization and optimization techniques, as well as their performance impacts of the medical image application on embedded multicore system. The results are encouraging to indicate that the high-level, productive programming model OpenMP can be adapted in the embedded multicore programming environment.

References

[1] M. Association *et al.*, "Multicore communications api specification version 1," *www. multicore-association. org*, 2008.

[2] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister, "Software standards for the multicore era," *Micro, IEEE*, vol. 29, no. 3, pp. 40–51, 2009.

[3] B. Meakin, "Multicore system design with xum: The extensible utah multicore project," Ph.D. dissertation, The University of Utah, 2010.

[4] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007, vol. 10.

[5] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," *Euro-Par 2012 Parallel Processing*, pp. 859–870, 2012.

[6] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in openmp," *A Practical Programming Model for the Multi-Core Era*, pp. 1–12, 2008.

[7] L. Meadows, "Experiments with wrf on intel® many integrated core (intel mic) architecture," *OpenMP in a Heterogeneous World*, pp. 130–139, 2012.

[8] I. Kozin, Science, and T. F. C. G. Britain), *Evaluation of clearspeed accelerators for hpc*. Science & Technology Faclities Council, 2009.

[9] M. Wolfe, "Implementing the pgi accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 43–50.

[10] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[11] L. Huang, B. Chapman, and C. Liao, "An implementation and evaluation of thread subteam for openmp extensions," in *Workshop on Programming Models for Ubiquitous Parallelism (PMUP 2006), Seattle, WA*. Citeseer, 2006.

[12] W. Jeun and S. Ha, "Effective openmp implementation and translation for multiprocessor system-on-chip without using os," in *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2007, pp. 44–49.

[13] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara, "Hierarchical parallelism control for multigrain parallel processing," *Languages and Compilers for Parallel Computing*, pp. 31–44, 2005.

[14] T. Weng and B. Chapman, "Implementing openmp using dataflow execution model for data locality and efficient parallel execution," in *Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-7)*.

[15] Z. Liu, B. Chapman, Y. Wen, L. Huang, T. Weng, and O. Hernandez, "Analyses for the translation of openmp codes into spmd style with array privatization," *OpenMP Shared Memory Parallel Programming*, pp. 26–41, 2003.

[16] L. Huang, G. Sethuraman, and B. Chapman, "Parallel data flow analysis for openmp programs," *A Practical Programming Model for the Multi-Core Era*, pp. 138–142, 2008.

[17] A. Eichenberger, O. Kathryn, K. Brien, P. Brien, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura *et al.*, "Optimizing compiler for the cell processor," 2005.

[18] 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, no. 4.5, pp. 589–604, 2005.

[19] D. Novillo, "Openmp and automatic parallelization in gcc," *Proceedings of the 2006 GCC Summit, Ottawa, Canada*, 2006.

[20] F. Liu and V. Chaudhary, "A practical openmp compiler for system on chips," *OpenMP Shared Memory Parallel Programming*, pp. 54–68, 2003.

[21] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku, "Evaluation of multicore processors for embedded systems by parallel benchmark program using openmp," *Evolving OpenMP in an Age of Extreme Parallelism*, pp. 15–27, 2009.

[22] H. Blume, J. Livonius, L. Rotenberg, T. Noll, H. Bothe, and J. Brakensiek, "Openmp-based parallelization on an mpcore multiprocessor platform-a performance and power analysis," *Journal of Systems Architecture*, vol. 54, no. 11, pp. 1019–1029, 2008.

[23] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of call statements," *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 176–185, 1986.

[24] W. Pugh, *The Omega test: a fast and practical integer programming algorithm for dependence analysis*. Citeseer, 1991.

[25] O. Hernandez, C. Liao, and B. Chapman, "Dragon: A static and dynamic tool for openmp," *Shared Memory Parallel Programming with Open MP*, pp. 53–66, 2005.

[26] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads Programming: A POXIS Standard for Better Multiprocessing*. O'Reilly, 1996.

[27] O. Van Bui, B. Chapman, R. Kufrin, D. Tafti, and P. Gopalkrishnan, "Towards an implementation of the openmp collector api," *Urbana*, vol. 51, p. 61801.

[28] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore, and F. Wolf, "Performance instrumentation and compiler optimizations for mpi/openmp applications," *OpenMP Shared Memory Parallel Programming*, pp. 267–278, 2008.

[29] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded

multicore MPSoC," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[30] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience, Special Issue on CPC'2006 selected papers*, 2006.

[31] D. Dart, "Dsp/bios kernel technical overview," *Texas Instruments, Application Report, Document no. SPRA780*, 2001.

[32] M. Ali, D. Magee, and U. Dasgupta, "Signal processing overview of ultrasound systems for medical imaging," *SPRAB12, Texas Instruments, Texas*, 2008.