High-Level Support for Pipeline Parallelism on Many-Core Architectures

Siegfried Benkner[1], Enes Bajrovic[1], Erich Marth[1], Martin Sandrieser[1], Raymond Namyst[2], and Samuel Thibault[2]

[1] Research Group Scientific Computing, University of Vienna, Austria
[2] University of Bordeaux, LaBRI-INRIA Bordeaux Sud-Ouest, Talence, France

Abstract. With the increasing architectural diversity of many-core architectures the challenges of parallel programming and code portability will sharply rise. The EU project PEPPHER addresses these issues with a component-based approach to application development on top of a task-parallel execution model. Central to this approach are multi-architectural components which encapsulate different implementation variants of application functionality tailored for different core types. An intelligent runtime system selects and dynamically schedules component implementation variants for efficient parallel execution on heterogeneous many-core architectures. On top of this model we have developed language, compiler and runtime support for a specific class of applications that can be expressed using the pipeline pattern. We propose C/C++ language annotations for specifying pipeline patterns and describe the associated compilation and runtime infrastructure. Experimental results indicate that with our high-level approach performance comparable to manual parallelization can be achieved.

1 Introduction

With the shift towards heterogeneous many-core architectures combining different types of execution units like conventional CPU cores, GPUs and other accelerators, the challenges of parallel programming will sharply rise. For an efficient utilization of such architectures usually different programming models and APIs, tailored for the different types of execution units, have to be combined within a single parallel application. Available technologies like Intel TBB [1], CUDA [2], Cell SDK [3], and OpenCL [4] are characterized by a low level of abstraction, forcing programmers to take into account a myriad of architecture details, usually beyond the capabilities of average users. Several recent research projects including ParLab [5], PetaBricks [6], Elastic Computing [7], ENCORE [8] and others, are addressing these challenges by proposing higher-level programming support for emerging many-core systems.

The European research project PEPPHER [9] targets programmability and performance portability for single-node heterogeneous many-core architectures by means of a component-based approach in combination with a task-parallel execution model. In this paper we present our contributions towards high-level

support for pipelined applications within the PEPPHER framework. Section 2 outlines the PEPPHER approach and describes the proposed language features for realizing pipelined C/C++ applications. Section 3 describes our source-to-source transformation framework as well as the coordination and runtime support for pipelining. Experimental results for two real-world applications and a comparison to TBB are presented in Section 4. The paper closes with a discussion of related work and future directions.

2 High-Level Programming Support

Since there exists no parallel programming model that covers all types of parallel applications and architectures, we argue that a programming framework for heterogeneous parallel architectures should support the use of different programming models and APIs within an application.

2.1 The PEPPHER Component Model

Within the PEPPHER model performance-critical parts of an application are realized by means of multi-architectural components that encapsulate, behind an interface, different implementation variants of a function[3] tailored for different execution units of a heterogeneous many-core system.

Component implementation variants are usually written by expert programmers using different programming APIs (e.g., CUDA, OpenCL) or are taken from optimized vendor-supplied libraries. Non-expert programmers may then construct applications at a high level of abstraction by invoking component functionality using conventional interfaces and source code annotations to delineate asynchronous (or synchronous) component calls. With this approach, a sequential program spawns component calls, which are then scheduled for task-parallel execution by the runtime system. A source-to-source compiler transforms annotated component calls such that they are registered with the runtime system and generates corresponding glue-code.

The compiler and runtime system make use of rich component meta-data, usually supplied by expert programmers via external XML descriptors. Besides information about the data read and written by components, meta-data includes information about resource requirements, possible target platforms, and performance relevant parameters [10]. The runtime system, built on top of the StarPU [11] heterogeneous runtime system, relies on a representation of the program as a directed acyclic graph (DAG) where nodes represent component calls (tasks) and edges represent data dependences. The runtime system dynamically schedules component calls to the available execution units of a heterogeneous many-core architecture such that (1) independent component calls execute in parallel on different execution units and (2) the "best" implementation variants for a given architecture are selected based on historical performance information captured in performance models.

[3] These functions must be pure, i.e. they must not access global data, they must be stateless, and they are to be executed non-preemptively.

2.2 Language Support for Expressing Pipeline Patterns

A pipeline consists of several inter-connected stages, where a stream of data flowing through the pipeline is processed at every stage. Data entering a stage via input port(s) is consumed, processed and emitted at the output port(s) accordingly. Usually, stages are connected via buffer structures from which data is fed into stages. While buffered pipelines require additional memory, they allow to decouple stages and mitigate relative performance differences. In our approach buffers between stages are generated automatically, but we provide language features for the user to control certain aspects of buffering.

The pipeline pattern has the potential of exploiting two levels of parallelism: inter-stage parallelism, if different stages execute on different cores, and, intra-stage parallelism, if a stage is itself parallel and executes on, e.g., a GPU.

In our framework, pipelines may be constructed from while loops where the loop body comprises two or more calls to multi-architectural components as considered within the PEPPHER framework. The *pipeline* pragma indicates that the subsequent *while* loop represents a pipeline. Each stage of the pipeline corresponds to a single component call statement within the loop body.

```
1 #pragma pph pipeline
2 while(data.size != 0) {
3   func1(iFile,data);              // connect func1 to func2 via data
4   #pragma pph stage replicate(4) // replicate stage 4 times
5   func2(data,cdata);              // connect func2 to func3 via cdata
6   func3(cdata,oFile);
7 }
```

Listing 1.1: Example of a pipeline directive.

2.3 Stage Replication and Stage Merging

Provided application logic permits, stage replication aims to increase the potential parallelism of pipelined applications by creating multiple replicas of a stage that may then be executed in parallel. Stages can be replicated using the *stage* pragma with the *replicate(N)* clause, specifying that a stage should be replicated N times (see Listing 1.1). As a consequence, multiple stage instances will be generated by the compiler to enable processing of different data packets in parallel, if enough execution units are available. While replication is a suitable technique for increasing pipeline throughput by replicating stages with (relative) high execution times, the programmer has to be aware that the order in which data-packets are processed may change (unless priority ordering is used), resulting in unpredictable application behavior. Moreover, sizes of connected in- and output buffers may have to be adapted as well. Also, stage replication might result in a performance degradation if not enough execution units are available to execute all stage replicas in parallel.

The *stage* pragma may also be used to merge multiple stages into a single stage (see, e.g., Listing 1.4). This allows the programmer to manually increase the granularity of stages, if the involved individual component calls do not exhibit enough computational density to mandate processing within a separate

stage. Note that stages can only be merged if for all involved stages compatible component implementation variants are available. The interface of the resulting single stage is automatically generated by the compiler, describing all input and output ports of the merged stage.

2.4 Buffer Management

With our framework buffers are automatically generated in between pipeline stages. These buffers temporarily store data packets, generated by the source stage(s), and consumed by the target stage(s). Depending on the type of application, different order guarantees and sizes for buffers may be required. Therefore we provide the buffer clause for specifying the order guarantee, including *priority*, *random*, and *fifo*, as well as the size of buffers (see Listing 1.2). Global buffer settings may be specified by using the *with buffer* clause together with the *pipeline* pragma. Local buffer settings for individual stages may be specified with the *buffer for port* clause within the *stage* pragma. In Listing 1.2 the buffer for port *cdata* in the second stage is changed to *RANDOM* and buffer size to 8, while for all other buffers *FIFO* ordering is used.

```
1 #pragma pph pipeline with buffer(FIFO)
2 while(data.size != 0) {
3   func1(iFile,data);
4   #pragma pph stage buffer for port(cdata,RANDOM,8)
5   func2(data,cdata);
6   func2(cdata,oFile);
7 }
```

Listing 1.2: Influencing Buffer Management.

3 Implementation

We have developed a prototype source-to-source compiler that transforms C/C++ applications with the described annotations into C++ with calls to a pipeline coordination layer that utilizes the StarPU [11] heterogeneous runtime system for scheduling stages for parallel execution onto the execution units of a heterogeneous many-core system comprising CPUs as well as GPUs. The source-to-source compiler has been implemented using the ROSE compiler infrastructure [12].

3.1 Source-To-Source Transformation

After the usual front-end processing phases an abstract syntax tree is constructed. Pipeline constructs are then further processed to determine the structure of the pipeline (stage interconnection) by analyzing the data types of objects passed between pipeline stages. For each stage interconnection (port) corresponding buffer structures, as specified globally or locally, are generated.

The generated target code contains calls to the pipeline coordination layer which comprises various classes for coordinating the execution of pipeline stages on top of the StarPU runtime system. The *Stage* class encapsulates information

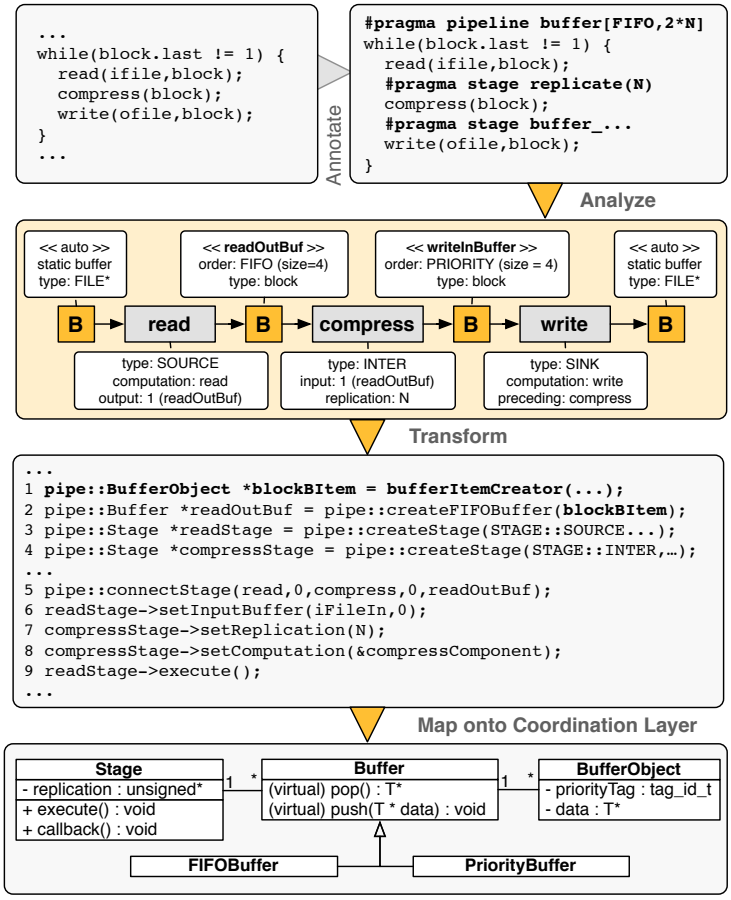

Fig. 1: Overview of the transformation process.

on the stage functionality (component), directly connected buffers and stages for each port, the number of instances of a stage to be processed in parallel (replication count), and the position of the stage within the pipeline (*source*, *inter*, or *sink*). Each stage owns a local coordination mechanism, described later, which slightly differs depending on the position of a stage. The abstract *Buffer* class generalizes the buffer access interface and is used to derive concrete buffer implementations like FIFO buffers or priority buffers. Each buffer stores information about connected stages, internal storage layout, and the type of data packets.

Figure 1 gives an overview for a three-stage data compression pipeline example. First buffer objects are generated (line 1) for holding data packets and meta-data such as priority tags, creation dates, and other information. Next, all inter-stage buffers are generated to hold the previously generated buffer objects,

setting the default data type for each buffer, as shown in line 2. Class instances for the *read* and *compress* stages are generated in line 3 and 4. The stages are further configured by specifying connected buffers, replication counts, and the stage computations, as shown in line 5 to 8. After all stages have been created and configured, stages are posted for execution to the runtime, initiating the actual execution of the pipeline, as shown in line 9.

3.2 Task-based Heterogeneous Runtime

The StarPU runtime system [11] utilized in our framework is based on an abstraction of the underlying heterogeneous many-core architecture as a set of *workers*, each representing either a general purpose computing core, or an accelerator (e.g., a GPU). The runtime system is responsible for selecting suitable component implementation variants for pipeline stages and for scheduling their execution to workers in a performance- and resource-efficient way, according to a specific scheduling policy. StarPU manages data transfers between workers, ensures memory coherency, and provides support for different scheduling strategies which may be selected at runtime.

The scheduling policy used within this paper is the Heterogeneous Earliest Finish Time [13] policy. This policy considers inter-component data dependencies, and schedules components to workers taking into account the current system load, available component implementation variants, and historical execution profiles, with the goal of minimizing overall execution time by favoring implementations variants with the lowest expected execution time.

3.3 Coordination

The coordination layer controls the execution of a pipelined application by deciding when to post stage component calls to the runtime system. We utilize a local coordination strategy where each stage is at runtime controlled by a corresponding stage object (an instance of the *Stage* class). The *Stage* class provides two methods for coordinating the execution of a pipelined application: the method *execute()* for posting a stage for execution to the runtime system and the method *callback()* for transferring control back to a stage object after its associated component has finished execution on a worker. In the following we outline a coordination scenario for the code shown in Figure 1.

First the runtime system is initialized and the required stage and buffer objects are instantiated. Next, the execution of all stages is initiated by invoking the *execute()* method on each stage object. This method posts a stage for execution to the runtime system provided its input buffer(s) and free slots in its output buffer are available. In our scenario, initially only the *read* stage is posted to the runtime while execution of all other stages is postponed since no input data is yet available. The runtime system then selects a suitable component implementation variant for the *read* stage and schedules it for execution onto a worker. When the *read* stage finishes execution on the selected worker, the

runtime system invokes its callback method. Within the callback, first all connected buffers of the *read* stage are updated and then the *execute()* method of the connected *compress* stage is called, which results in posting the *compress* stage to the runtime system. Finally, if a new data packet to be processed and a free output buffer slot are available, the *read* stage re-posts itself for execution to the runtime system. The runtime can now schedule the second instance of the *read* stage and the first instance of the *compress* stage for parallel execution on different workers. With this scheme, stages coordinate themselves only in combination with their immediate neighbors, but without a central coordinator.

4 Experimental Evaluation

An initial evaluation has been performed with two real world codes, a data compression application and a face detection application. We compare our approach to Threading Building Blocks (TBB) as well as to existing parallel implementations on two different architectures. Architecture A represents a homogeneous system with two Intel Xeon X7560 (8 cores, 2.26 GHz) running RHEL 5. Architecture B is a heterogeneous system with two Intel Xeon X5550 (4 cores, 2.67 GHz), one Nvidia GeForce GTX 480 (480 Cores, 1.5GB, 1.40GHz), and one Nvidia GeForce GTX 285 (240 Cores, 1GB, 1.48GHz), CUDA 4.0 and RHEL 5.6. The performance numbers shown are average execution times over ten runs.

4.1 BZIP2 Compression

BZIP2 is an open-source lossless data compression tool, based on the Burrows-Wheeler-Transformation [14]. Compared to other compression techniques, BZIP2 is embarrassingly parallel, compressing data at the granularity of blocks with a fixed size. Since there are currently no heterogeneous implementations available, BZIP2 cannot utilize the GPUs of architecture B.

Listing 1.3 outlines the implementation using our language constructs. We implement a three stage pipeline with *FIFO* and *PRIORITY* buffers and utilize the replication feature for the middle stage.

```
1 unsigned int N = get_max_cpu_cores();
2 #pragma pph pipeline with buffer(FIFO,N*2)
3 while(b != 0) {
4   readBlock(file,b);
5   #pragma pph stage replicate(N)
6   compress(b);
7   #pragma pph stage buffer for port(write.b,PRIORITY)
8   writeCompressedBlock(file,b);
9 }
```

Listing 1.3: BZIP2 Compression Pipeline

For comparison, we have implemented bzip2 using the pipeline pattern of TBB to create a pipeline [15] with three stages (read, compress, write) using the utility functions provided by the bzip2 library [16]. Since in TBB stages cannot be replicated explicitly, we have set the number of alive objects to twice

	Architecture A					Architecture B			
# Cores	1	2	4	8	16	1	2	4	8
TBB	248.26	135.63	66.72	32.12	14.44	245.52	122.01	62.75	33.19
pbzip2	250.31	130.01	65.80	32.94	19.38	248.81	131.33	65.72	36.96
PEPPHER	245.40	141.68	75.62	35.42	16.83	238.47	109.87	61.97	32.83

Table 1: bzip2 Performance Results (execution times in s)

the number of cores available. This allows TBB to schedule stages in parallel if possible. For correct compression in time, we have enabled order preservation. Moreover, we measured an existing code from the pbzip2 project [17], which represents a manually implemented pipeline with three stages (read, compress, and write) and priority buffers.

Table 1 shows the measured (average) execution times for the bzip2 benchmark for compressing a file with size of 1 GB. As can be seen, our high-level approach delivers performance results which are very similar to the other two approaches. However, the programming effort with our approach is significantly reduced, since our version requires only 80 lines of code, while pbzip2 comprises approx. 4000 lines of code and the TBB version 154 lines of code.

4.2 OpenCV Image Processing

The Open Source Computer Vision (OpenCV) library provides extensive support for the implementation of real time computer vision applications [18]. Originally developed for homogeneous architectures, current releases offer built-in support for GPUs based on CUDA. Using OpenCV we have implemented a face detection application in a pipelined manner, where for the detection stage two different implementation variants, for CPUs and GPUs, are provided.

```
1  unsigned int N = get_max_execution_units();
2  #pragma pph pipeline with buffer(PRIORITY,N*2)
3  while(image.number < 32) {
4    readImage(file,image);
5    #pragma pph stage replicate(N) {
6       resizeAndColorConvert(image);
7       detectFace(image,outImage);
8    }
9    writeFaceDetectedImage(file,outImage);
10 }
```

Listing 1.4: OpenCV Image Processing Pipeline

Listing 1.4 sketches our implementation as a three-stage pipeline with priority buffers. The pipelines exits once 32 images have been processed. The middle stage, which merges two component calls, is replicated according to the available number of execution units within the system.

For comparison, a pipelined TBB version [15] has been implemented in a similar way. Again, instead of stage replication, we have set the number of alive data packets to twice the number of execution units, enabling TBB to schedule stages in parallel.

Architecture A				
Image Size	VGA	SVGA	XGA	QXGA
TBB (1 Core)	15.61	23.51	41.84	170.58
PEPPHER (1 Core)	12.40	17.85	30.72	140.86
Intel TBB (16 Core)	1.26	1.92	3.39	13.60
PEPPHER (16 Cores)	1.16	1.72	2.91	12.33
Architecture B				
TBB (1 Core)	12.75	20.07	35.15	145.68
PEPPHER (1 Core)	9.62	14.33	24.94	111.45
PEPPHER (1 Core + 1 GPU)	3.94	5.91	10.35	46.30
PEPPHER (1 Core + 2 GPUs)	2.95	2.72	6.53	30.81
TBB (8 Cores)	1.47	2.29	4.13	17.4
PEPPHER (8 Cores)	1.18	1.78	3.58	13.69
PEPPHER (7 Cores + 1 GPU)	1.13	1.63	2.91	11.89
PEPPHER (6 Cores + 2 GPUs)	0.94	1.40	2.44	10.71

Table 2: OpenCV Performance Results (execution times in secs)

Table 2 shows performance results for the face detection code in comparison to the TBB version using different image resolutions including VGA(640x480), SVGA(800x600), XGA(1024x768), and QXGA(2048x1536). As can be seen, on architecture A our high-level approach outperforms the TBB version by about 20%. On architecture B, when using only the CPU cores, we get similar results. As opposed to TBB, however, our approach can take advantage of the GPUs by utilizing the GPU implementation variant for the (merged) middle pipeline stage. Since for each GPU an additional CPU core is required, the number of usable general purpose cores is reduced accordingly. As can be seen, using one CPU core and one GPU (GTX 460) the execution time is reduced by a factor of up to 3.14 compared to the TBB version using one CPU core. Using a second GPU results in only modest further performance improvements. Again the results show that our high-level approach to pipelining has the potential to outperform TBB, while significantly improving programmability. Moreover, based on our concept of multi-architectural components together with a versatile heterogeneous runtime system, we can take advantage of a heterogeneous CPU/GPU-based architecture without modifying the high-level application code.

5 Related Work

Over the last two decades design patterns as well as skeletons had a significant impact on software development in general as well as on parallel and distributed programming [19–22].

Intel Threading Building Blocks (TBB) provides direct support for pipeline patterns. As opposed to our work, TBB targets only homogeneous architectures and does not support stage implementation variants.

Thies at al. [23] propose StreamIt, a domain specific language (DSL) for designing stream-based applications where in combination with the underlying compiler pipeline specific optimizations [24] are applied. However, support for heterogeneous architectures is not addressed in this work.

Schaefer at al. [25, 26], propose a language-based approach for engineering parallel application using tunable patterns. Although different stage implementations are supported within the pipeline pattern, support for heterogeneous architectures has not been addressed.

Suleman at al. [27], propose a feedback-directed approach with integrated support for tuning. In combination with online-monitoring, pipelined applications are optimized on a coarse-grain level. Neither different stage implementation variants, nor optimizations for heterogeneous architectures are supported.

There have been several proposals for extending C or Fortran in order to support programming of heterogeneous systems comprised of CPUs and GPUs including OmpSS [28], PGI Accelerate [29], HMPP [30], and OpenACC[31]. These approaches are based on directives for specifying regions of code in Fortran or C programs that can be offloaded from a host CPU to an attached GPU by the compiler. None of these approaches, however, supports pipelining or other parallel patterns.

6 Conclusion and Future Work

We have presented high-level language annotations for C/C++ for developing pipelined applications on heterogeneous many-core architectures without having to deal with complex low-level architectural issues. We provided an overview of the associated implementation framework which is currently being developed within the European PEPPHER project. Our work relies on a component-based approach where pipeline stages correspond to multi-architectural components that encapsulate different implementation variants optimized by expert programmers for different execution units of a heterogeneous many-core system. A source-to-source compiler translates pipelined applications to an object-oriented coordination layer which is built on top of a heterogeneous task-based runtime system. The task-based runtime system attempts to schedule the best component implementation variants for parallel execution on the free execution units of a many-core system such that overall performance is optimized.

Our approach enables to run the same high-level application code without changes on homogeneous and heterogeneous multi-core architectures, delegating to the runtime system task scheduling and implementation variant selection. Experimental results on two different architectures are encouraging and indicate that a performance comparable to manual parallelization can be achieved, despite the considerably higher level of abstraction provided by our language features for the pipeline pattern.

For future work we plan to extend our framework with autotuning capabilities to, for example, determine replication counts and buffer sizes for pipeline stages. Furthermore, we plan to experiment with different runtime scheduling strategies as supported by StarPU and to provide support for other parallel patterns.

Acknowledgment This work was supported by the European Commission as part of the FP7 Project PEPPHER under grant 248481.

References

1. Intel, "Threading Building Blocks," 2009. [Online]. Available: http://threadingbuildingblocks.org
2. C. Nvidia, *Compute Unified Device Architecture Programming Guide*. NVIDIA: Santa Clara, 2007.
3. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. J. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4–5, pp. 589–604, 2005.
4. A. Munshi (ed.), *OpenCL 1.0 Specification*. Khronos OpenCL Working Group, 2011.
5. H. Pan, B. Hindman, and K. Asanović, "Composing Parallel Software Efficiently with Lithe," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 376–387.
6. J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, 2009, pp. 38–49.
7. J. R. Wernsing and G. Stitt, "Elastic Computing: A Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2010, pp. 115–124.
8. H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel Programming of General-Purpose Programs using Task-based Programming Models," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'11, Berkeley, CA, USA, 2011, pp. 13–13.
9. S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," *Micro, IEEE*, vol. 31, no. 5, pp. 28 –41, sept.-oct. 2011.
10. M. Sandrieser, S. Benkner, and S. Pllana, "Using explicit platform descriptions to support programming of heterogeneous many-core systems," *Parallel Computing*, vol. 38, no. 12, pp. 52 – 65, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001396
11. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures." *Concurrency and Computation: Practice and Experience*, no. 23, pp. 187–198, 2011.
12. D. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 49, 2005.

13. H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, Mar 2002.
14. M. Burrows, "A Block-Sorting Lossless Data Compression Algorithm." Research Report 124, Digital Systems Research Center, 1994.
15. Intel, "Intel Threading Building Blocks - Pipeline Documentation." [Online]. Available: http://threadingbuildingblocks.org/files/documentation/a00150.html
16. J. Seward, "BZIP2 Library Utility Function Documentation," 09 2011. [Online]. Available: http://bzip.org/1.0.5/bzip2-manual-1.0.5.html#util-fns
17. J. Gilchrist, "Parallel Data Compression with bzip2," *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, vol. 16, pp. 559–564, 2004.
18. B. Gary, *Learning openCV: Computer Vision with the openCV Library*. O'Reilly USA, 2008.
19. A. Benoit and Y. Robert, "Mapping Pipeline Skeletons onto Heterogeneous Platforms," *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007*, May 2007.
20. M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Computing*, 2004.
21. T. Mattson, B. Sanders, and B. Massingill, "Patterns for Parallel Programming," Addison-Wesley, 2005.
22. A. Pop and A. Cohen, "A Stream-Computing Extension to OpenMP," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 2011.
23. W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, Apr. 2002.
24. J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache Aware Optimization of Stream Programs," *ACM SIGPLAN Notices*, vol. 40, no. 7, 2005.
25. C. Schaefer, V. Pankratius, and W. Tichy, "Engineering Parallel Applications with Tunable Architectures," *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1, May 2010.
26. F. Otto, C. Schaefer, M. Dempe, and W. Tichy, "A Language-Based Tuning Mechanism for Task and Pipeline Parallelism," *Euro-Par'10: Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Aug. 2010.
27. M. Suleman, M. Qureshi, Khubaib, and Y. Patt, "Feedback-Directed Pipeline Parallelism," *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.
28. E. Ayguadé, R. M. Badia, D. Cabrera, A. Duran, M. González, F. D. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Pérez, and E. S. Quintana-Ortí, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures," in *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP (IWOMP)*, vol. 5568, 2009, pp. 154–167.
29. M. Wolfe, "Implementing the PGI Accelerator Model," in *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, Mar. 2010.
30. F. Bodin and S. Bihan, "Heterogeneous Multicore Parallel Programming for Graphics Processing Units," *Scientific Programming*, vol. 17, pp. 325–335, 2009.
31. "OpenACC. Directives for Accelerators." [Online]. Available: http://www.openacc-standard.org/