Autotuning of Pattern Runtimes for Accelerated Parallel Systems

Enes BAJROVIC, Siegfried BENKNER [1], Jiri DOKULIL,
Martin SANDRIESER
Research Group Scientific Computing
University of Vienna, Austria

Abstract. Parallel architectures with node-level accelerators promise significant performance improvements over conventional homogeneous systems. To cope with the increased complexity of programming such systems various pattern-based programming libraries have become available. In this paper we present our work on providing autotuning capabilities for two runtime libraries that provide parallel programming patterns on state-of-the-art heterogeneous hardware. We present a brief overview of these runtime libraries, outline possible integration with existing tuning frameworks and present initial experimental results.

Keywords. Heterogeneous many-core architectures, accelerators, parallel patterns, autotuning, runtime libraries.

Introduction

Parallel computing systems featuring node-level accelerators are increasingly being used for high-performance computing applications since such systems often deliver significant performance improvements compared to conventional homogeneous systems. However, the increased heterogeneity resulting from the utilization of different processing units makes efficient programming of such architectures a challenging task. One established way to tackle the high programming complexity of parallel systems is the use of well-defined parallel programming patterns for application development. This methodology has already been applied successfully by various library-based approaches targeting accelerated parallel systems (e.g., Thrust [1], SkePU [2], HyPHI [3]). However, even though pattern-based programming libraries may significantly improve programmability for the end-user, the portable and efficient implementation of pattern libraries themselves is very demanding. In order to achieve good performance with such libraries on different heterogeneous systems, multiple application or platform-specific decisions must be made that may require automatic tuning support.

In this paper, we present our work on providing autotuning capabilities for two runtime libraries that provide parallel programming patterns on state-of-the-art heterogeneous hardware. We present a brief overview of these runtime libraries and outline possible integration with existing tuning frameworks.

[1] Corresponding Author: siegfried.benkner@univie.ac.at

1. HyPHI Library

The HyPHI library[3] is a C++ template library that provides high-level parallel patterns that are executed in a hybrid fashion on the host and (possibly several) Xeon Phi coprocessors simultaneously. An example pattern is hybrid for-each, that applies a function to every item in a container, processing some items on the host and some on a coprocessor. The following code example quickly demonstrates the way the library is used to apply a functor to each element in a sequence.

```
struct functor {
  int factor;
  functor(int f) { factor=f; }
  void operator()(int& x) const { x=x*factor; }
};

std::vector<int> data;
fill_data(data);
functor f(2);
hybrid_for_each(data.begin(),data.end(),f);
```

The hybrid execution model used by HyPHI has important implications on the autotuning process. We have to address the issue that the library splits the work between two machines with different characteristics. The Phi has different vector units (wider, different instruction set), higher number of cores (and 4-way hyper-threading), and different memory architecture. Both systems provide fully cache coherent random memory access, but the details differ significantly. For example, the host will probably be a NUMA machine, while the Phi uses a ring bus to connect the cores to the memory. As a result, executing the same operation on the host and on the coprocessor can result in very different performance. In extreme cases, the difference can be over an order of magnitude.

1.1. Tuning Parameters

A very important factor from the tuning point of view is how work is distributed among host processors and coprocessors. With the HyPHI library work distribution is performed at runtime depending on five parameters: (1) the host batch size, which specifies the number of items allocated to the host in one batch, (2/3) the minimal/maximal coprocessor batch size, which specifies the minimal/maximal number of items allocated to a coprocessor in one batch, (4) the coprocessor buffer count, which is the number (per coprocessor) of buffers used for data offloading, and (5) the coprocessor count, which designates the number of coprocessors to be used during execution.

Choosing these parameters such that overall execution time is minimized depends on the properties of the function provided by the user of the library and possibly also the values of the data, so it is usually not possible to determine the optimum at compile time. For example, if the workload runs very well on the coprocessor and the data set contains a large number of items, it is best to use a large batch for the coprocessor, multiple (usually 3-5) buffers and all available coprocessors. However, if the data set contains just a few items, it may be best to reduce the overhead to a minimum and not use any coprocessors, i.e.,

completely switch off data offloading. Or, if the coprocessor is not that efficient at processing a workload, offloading smaller batches may allow the library to better use the power of the host near the end of the execution, providing better overall performance. As a result, we tune each combination of algorithm, functor and data size (rounded up to the nearest power of two) individually as completely different entities.

We have designed the HyPHI library in such a way that all tuning parameters may be changed "on the fly" – a running HyPHI algorithm can be reconfigured at any time by providing a new set of parameter values. The new values take effect as soon as possible. For example, if the buffer size parameter is increased, then the next time a buffer is offloaded it uses the new size. This allows us to implement a watchdog that observes the library and if a set of tuning parameters turns out to be especially bad, it can change the parameters to a different set.

Another reason why dynamic reconfiguration is important for autotuning is the fact that information used as input for the tuning process (like workload characteristics or performance of previous executions) or the tuning module may not always be available. We use a separate process (in the operating system sense) to make the tuning decisions. This process is called *agent*. This makes it easier to include HyPHI in an application, since the library has almost no external dependencies when done like this. Also the agent can track all running HyPHI instances and take concurrent execution into consideration. However, the agent may not always be available, it may take too long to respond, or it may not have enough data (yet) to use as an input for the tuning.

1.2. Tuning Strategies

To deal with the fact that tuning data or decision may not be available, we have implemented multiple tuning strategies that form a sequence from the simplest to the most sophisticated. If a higher-level tuner is unable to make a decision (provide a set of tuning parameter values), it is possible to fall back to a simpler tuner until the lowest level, which is always able to make a decision. The following sections describe the individual levels.

Default The simplest strategy is to provide a fixed default set of parameter values. These values have been selected based on our previous experiments to provide reasonable performance in most cases, except for the most pathological ones. However, the performance is usually not optimal and a better set of values may provide significant performance increases.

Cached values It is possible to store a set of values locally on disc in a file. These values are a result produced by a higher-level tuner. The tuner must explicitly report the values as being "the best" (rather than an experimental set of values) to have them stored in the file. Currently, only the highest-level tuner uses experimental sets. Other tuners provide only "the best" values.

Observers and heuristics While the HyPHI algorithm is executing, simple logging mechanisms are used to observe the properties of the current workload. Currently, the average times needed by the host/coprocessor to execute the functor (process a single item) are logged. These observations are sent to the host even

while the execution is still running. Once the algorithm finishes, the aggregated results are stored in a file. If the file already exists, the new results are combined with the older data to provide average values.

The tuning decisions are made by a simple heuristic that classifies the workload into one of predefined classes (currently three classes are used) and returns a set of tuning parameter values that should work with the selected class. The values are hard-coded for each class. The classification uses observed characteristics of the workload as its input. If there is a set of historical observations available, they are used at the start of the execution of a HyPHI algorithm to classify the workload and configure the parameters. If no historical data is available, the tuner waits until real-time observer data for the host and coprocessors becomes available and makes the decision on that data. It may take some time for the coprocessor data to become available, since it is sent to the host piggybacked to the processed data, so it is only available after an offloaded batch has been processed, re-serialized and sent back to the host. After the workload has been classified, no further tuning decisions are made – reconfiguration is not used by this tuner.

Tuning agent This is the highest-level tuner. It connects to the tuning agent (separate process that provides tuning decisions to all HyPHI instances), requests a set of tuning parameter values, and uses the set. Then, it waits on the connection, in case a different set of values gets sent by the agent. In that case, the HyPHI parameters get updated to the new set – dynamic reconfiguration takes place. At the end of the execution, the tuner reports the total execution time to the agent. All decisions take place in the agent, the tuner only provides the basic data and forwards the tuning decisions to the runtime.

The agent is in fact a network server that listens for incoming connections from the HyPHI instances. Each instance sends over the identification of the running workload (algorithm, functor, and data size). The HyPHI uses the SPOT [4] tuning library to first generate an initial set of experiments – sets of parameter values. The agent then uses the experiments and sends the values to consecutive instances of the corresponding HyPHI workload. For each set, the instance returns the total running time. These times are stored and after all experiments have finished, they are sent to the SPOT library. The library builds a model based on these data and creates a new set of experiments that explore the promising parts of the model in greater detail. As a result, after the initial experiment set is finished, the agent tends to only send out configurations that provide good performance – there are no further "bad tuning decisions".

This setup also allows the agent to see, whether multiple HyPHI algorithms run concurrently on the same machine or not. At the moment, we use this information to reject performance results returned by HyPHI instances that executed concurrently with one another.

2. PEPPHER Pipeline Coordination Library

In the context of the European PEPPHER project [5], we have developed language, compiler and runtime support for optimizing pipeline patterns for heterogeneous many-core architectures [6]. Within the PEPPHER framework, pipelines

are realized based on while-loops with source-code annotations. Pipeline stages usually correspond to calls to multi-architectural components, for which multiple implementation variants may be provided. Such component implementation variants may be optimized for different execution units of a heterogeneous target architecture, e.g., for a homogeneous multi-core CPU, for a GPU, or for other types of accelerators.

An example of a high-level C++ pipeline code for face detection in a stream of images is shown below. The first pipeline stage reads images from an input file, the middle stages perform image transformation and face detection by means of calls to multi-architectural PEPPHER components, and the last stage writes an image with all detected faces marked with rectangles to an output file. For the two middle stages, two different component implementation variants are provided within the PEPPHER framework, one optimized for execution on a conventional CPU core and one optimized for GPUs. These implementation variants have been re-engineered from the OpenCV image processing library. By means of annotations, the user can specify what kind and size of buffers should be generated for passing data between pipeline stages. Moreover, the user can specify a replication factor for individual pipeline stages in order to influence the degree of parallelism during execution.

```
#pragma pph pipeline with buffer (UNORDERED, N*2)
while ( inputstream >> file ) {
  ReadImage(file, image);
  ResizeAndColorConvert (image, outimage);
  #pragma pph stage replication(rfactor)
  DetectFace(outimage);
  #pragma pph stage with buffer (PRIORITY, N*2)
  WriteFaceDetectedImage(file, outimage);
}
```

This annotated high-level code is then transformed by a source-to-source compiler into code that utilizes a pattern coordination runtime library for parallel execution on heterogeneous many-core architectures. The pattern coordination library manages all aspects of execution on a heterogeneous many-core architecture, including the automatic management of buffers for data passed between pipeline stages, the replication of individual stages, and the coordination of task-parallel execution of pipeline stages. Internally, the pipeline coordination library utilizes the StarPU [7] heterogeneous runtime system, which is responsible for dynamically selecting suitable component implementation variants for pipeline stages and for scheduling their execution to the different execution units of a heterogeneous many-core system in a performance- and resource-efficient way. StarPU also manages data transfers between execution units, and provides support for different scheduling strategies, with the goal of utilizing all execution units of the target architecture.

Within the AutoTune project our goal is to develop autotuning techniques for optimizing such high-level pipeline applications for CPU/GPU-based systems. For this purpose, the pipeline coordination library exposes different parameters that are amenable to autotuning.

2.1. Tuning Parameters

The pipeline coordination library enables dynamic reconfiguration by exposing a set of tuning parameters, thus allowing users or external tools to tune the execution of the pipeline in order to achieve a desired goal (e.g., to maximize pipeline throughput). Currently we support the following tuning parameters: (1) the stage replication factor, which determines the number of stage instances that may be executed in parallel, (2) the sizes of buffers to hold data packets passed between pipeline stages, (3) the number of CPU cores and (4) the number of GPUS to be used, and (5) the scheduling strategy used by StarPU for scheduling component calls to free execution units of the target system.

All these parameters have a profound influence on the performance of applications that rely on pipeline patterns. Finding the best parameter combination for a given application, problem size, and machine configuration is a challenging task that we tackle in the context of the AutoTune project with the Periscope Tuning Framework (PTF) [8].

2.2. Integration with Periscope Tuning Framework

The Periscope Tuning Framework (PTF) [8] is an extension of the Periscope online performance analysis tool [9]. PTF aims at providing an infrastructure for automated code-tuning based on expert knowledge about performance characteristics of target applications and computational patterns. Hence, it features an extensible architecture based on *tuning-plugins* that enable external developers to guide tuning decisions. For evaluation of possible tuning variants, the framework provides highly-distributed monitoring and measurement facilities enabling the detailed assessment of runtime-specific performance characteristics.

Instead of integrating autotuning functionality into pattern-based libraries solely internally, we aim at implementing a generic mechanism for utilization of external tuning frameworks such as PTF. Since tuning objectives and decisions should be adaptable and might change depending on target hardware or application requirements, we believe that this approach can provide the required flexibility. For example, our approach could be combined with other PTF plugins or a variety of different external tuning frameworks and optimization strategies. In the following we outline the steps required for integrating our pipeline pattern runtime library with the Periscope Tuning Framework:

(1) Exposure of metrics – To evaluate tuning objectives, several library- and application-specific performance characteristics need to be defined within PTF and measured. Such metrics can be pattern-specific (e.g., buffer occupancy of pipeline stages) or generic (e.g., wall-clock time). The Periscope framework already provides a wide variety of different metrics which are organized in measurement groups. New metrics (e.g., pipeline-stage execution time) representing measurements done by the pipeline coordination layer were added to the appropriate Periscope source files.

(2) Definition of performance properties – Based on the measured metrics, performance properties indicate relevant performance characteristics of an application or library under investigation. For each computational pattern provided

by the runtime library, performance properties with high significance regarding the tuning objective should exist. Therefore, we added performance properties relevant for our pipeline coordination library. Such property classes have been derived from a *Property* class available in the Periscope framework and were added to the Periscope build system.

(3) Runtime measurements – The Periscope framework is based on the idea of *selective* monitoring. This aims at reducing the amount of acquired data for an experiment. Therefore, the framework employs a *Monitoring Request Interface* (MRI) used to retrieve information. This mechanism is based on instrumentation of the investigated application using a hierarchical identification scheme of *program regions*. [10]

To enable the measurement of pipeline specific metrics, we added special *pipeline region* identifiers to Periscope. In addition, we extended the pipeline coordination layer with functions to start/stop performance measurements and to deliver the acquired data to the Periscope framework on it's request. The support for pipeline region measurements can be enabled at compile time.

(4) PTF tuning plugin development – For each tuning case, a PTF tuning plugin needs to be provided that guides the search for good tuning variants (i.e., concrete set of tuning parameters). This search may incorporate knowledge about problem-, platform- or pattern-specific properties that may help to prune the search space or find good solutions in reasonable time. For a first prototypical implementation, we developed a basic tuning plugin that employs an exhaustive search. The plugin uses the execution time of a pipeline region, that was previously measured by the coordination layer (see (3)), to evaluate performance and find good tuning decisions.

(5) Tuning parameter setting – Actual tuning variants for execution (i.e., concrete set of tuning parameters) must be communicated to all library instances. The PTF provides routines for dynamically setting values of tuning variables as well as for calling special tuning functions. Hence, the pipeline coordination layer supports reconfiguration at runtime. In the current implementation, reconfiguration is based on integer variables whose values are set at runtime by the PTF. This happens before the according reconfiguration calls to the pipeline coordination library.

3. Experiments

3.1. HyPHI Library

The results for the HyPHI library were obtained on a server machine with four Intel Xeon Phi 5110P coprocessors (60 cores, 1.053GHz, 8GB RAM). The machine itself had two E5-2650 CPUs (2GHz, 20MB cache) and 128GB RAM. The experiment executed an artificial workload that performed identical work on a sequence of 50 000 items. We have performed one test with the agent and the SPOT library [4] and one test that did an exhaustive search of the parameter space. The SPOT library was set up to use Latin hypercube for the initial design (10 configurations, each to be executed 2 times) and random forest as the prediction model. The exhaustive search explored 33788 possible configurations.

The exhaustive search found the best configuration to provide the run time of 2.19754 seconds. With three exceptions, the run times were below 10 seconds. All run times add up to 98404 seconds (27.33 hours).

With SPOT, the initial set of experiments (the initial design) contains some sets of parameter values that are far from ideal, resulting in a long execution time. In our experiment, that meant almost 5 times longer than the best time, similar to the worst options discovered by the exhaustive search. However, the initial experiments did provide enough data to make the next set (5 configurations, each with 2 repetitions) perform much better, with runtimes less than double of the overall best. In the following iterations, all run times were within 20% of the best time. After four sets of experiments (total of 50 executions of the HyPHI algorithm), the average time of the best configuration discovered by SPOT was 2.24007. The number is an average of four executions of that configuration.

An important factor that comes into play with these numbers is the (in)stability of the HyPHI execution time. Due to complex scheduling and work distribution used by the library, the execution times generally vary by around 10%, with rare outliers being much further away (generally within double of the average value). Second important factor is the fact, that the performance of the library is "stable" with regards to the configuration, which means that small change of the configuration (especially the batch and buffer sizes) has no or very little effect on the performance. Last thing to note is that SPOT always requires each configuration to be executed at least twice. The number of repetitions is provided by the library and it also sometimes requests further experiments for an already tested configuration. This happens if the configuration looks promising, in order to verify that the good result was not caused by random fluctuation.

These factors together explain our experimental results, where we had 4 executions of the best option found by SPOT and the average time was longer than the best performer found by the exhaustive search. After executing the configuration found by the exhaustive search 4 times, the average time was 2.30539 seconds, much worse than the one found by SPOT. Due to the nature of the HyPHI library, the SPOT is able to quickly eliminate the bad options and after one or two iterations, it ends up just fine-tuning the buffer sizes.

3.2. Pipeline Coordination Library

For demonstration we use the face detection example described in Section 2. The application processes a set of 350 images of nHD (640x360) resolution, each containing an arbitrary number of human faces. Performance measurements were performed on a machine with two quad-core Intel Xeon X5550 CPUs (2.66GHz, 24GB RAM) and NVIDIA Tesla C2050 and C1060 GPUs, respectively.

Using the PTF tuning plugin, we used exhaustive search to find the best configuration for the available tuning parameters. We considered structural properties of pipelines and runtime parameters. This resulted in five tuning points (1) stage replication factor of the most performance demanding stage (DetectFace()), (2) size of the input buffer of the DetectFace() stage, (3) number of CPU cores, (4) number of GPUs, and (5) the scheduling policy - EAGER (simple greedy scheduler) versus HEFT (Heterogeneous Earliest Finish Time) [7]. In total PTF

explored 360 possible configurations, spending almost 6.2 hours in doing so. In Table 1, we summarize the explored values for each tuning parameter.

Tuning Parameter	**Possible values**	**Best Configuration Variant**
Replication Factor	1, 2, 4, 8	8
Number of CPU cores in use	1, 2, 4, 6, 8	6
Number of GPUs in use	0, 1, 2	2
Scheduling Policy	"eager", "heft"	"heft"
Buffer Size	1, 8, 16	8

Table 1. Possible values of tuning parameters.

The desired outcome is the maximization of the pipeline throughput, which translates to minimization of the overall execution time. The best configuration set performed face detection over the complete data set in 9.1 seconds. It used replication factor of 8, 6 CPU cores, 2 GPUs, buffer size of 8 and HEFT scheduling policy. The slowest configuration set utilized only 1 CPU core, which resulted in execution time of 95.4 seconds. The slowest configuration that utilized the whole system resulted in the execution time of 20.2 seconds.

4. Related Work

Existing autotuning efforts include (1) self-tuning specialized libraries (e.g., linear algebra or signal processing) like ATLAS[11] or FFTW[12], (2) tools that automatically search for best combination of compiler optimization parameters [13,14], and (3) tools that search for best values of application-level parameters [15,16]. Even though both of our solutions are in fact libraries and the tuning parameters control the behavior of those libraries, our work is closer to the third group (application tuning) than the first one (self-tuning libraries), because we execute computational kernels that are not part of the library and can behave very differently from each other. Our efforts are close to the emerging area of (possibly automated) tuning of OpenCL or CUDA parameters [17,18].

5. Conclusion and Future Work

In this paper we presented our ongoing work of autotuning support for pattern-based runtime libraries for programming heterogeneous many-core architectures. Currently, the HyPHI library and the pipeline coordination library use different autotuning facilities. We plan to integrate both libraries with the Periscope Tuning Framework, which is currently under development in the European AutoTune project, and investigate the use of more intelligent search strategies.

Acknowledgment

This work was supported by the European Commission's FP7 project AutoTune under grant no. 288038 (see http://www.autotune-project.eu).

References

[1] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems, Jade Edition* (W. mei Hwu, ed.), Morgan Kaufmann, 2011.

[2] J. Enmyren and C. W. Kessler, "Skepu: a multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, (New York, USA), ACM, 2010.

[3] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer, "HyPHI – task based hybrid execution C++ library for the Intel Xeon Phi coprocessor," in *42nd International Conference on Parallel Processing (ICPP-2013), Lyon, France*, 2013.

[4] T. Bartz-Beielstein, "SPOT: An R Package For Automatic and Interactive Tuning of Optimization Algorithms by Sequential Parameter Optimization," *CoRR*, vol. abs/1006.4645, 2010.

[5] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, 2011.

[6] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, "High-Level Support for Pipeline Parallelism on Manycore Architectures," in *Euro-Par 2012 Parallel Processing - 18th International Conference*, vol. 7484, pp. 614–625, 2012.

[7] 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, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[8] R. Miceli, G. Civario, A. Sikora, E. Csar, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications," in *Applied Parallel and Scientific Computing* (P. Manninen and P. ster, eds.), vol. 7782 of *LNCS*, pp. 328–342, Springer, 2013.

[9] S. Benedict, V. Petkov, and M. Gerndt, "PERISCOPE: An Online-Based Distributed Performance Analysis Tool," in *Tools for High Performance Computing 2009* (M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel, eds.), pp. 1–16, Springer, 2010.

[10] M. Gerndt and E. Kereku, "Selective instrumentation and monitoring," in *proceedings of 11th workshop on compilers for parallel computers (CPC 04), Kloster Seeon*, pp. 61–74, 2004.

[11] C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS Project," *PARALLEL COMPUTING*, vol. 27, 2001.

[12] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384, 1998.

[13] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, "Compiler optimization-space exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 204–215, 2003.

[14] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic selection of compiler options using non-parametric inferential statistics," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 123–132, 2005.

[15] I.-H. Chung and J. K. Hollingsworth, "Using information from prior runs to improve automated tuning systems," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, 2004.

[16] Y. L. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman, "Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization," in *International Parallel and Distributed Processing Symposium*, pp. 1–8, 2008.

[17] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 195–204, 2008.

[18] Y. Liu, E. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *International Parallel and Distributed Processing Symposium*, 2009.