School of Informatics

Informatics Research Review (INFR11136)
Machine Learning For Program Optimizations

s1884908 (Exam Number: B134199)
January 2019

Abstract

Machine learning provides a large number of algorithms that help reduce manual effort and improve the performance of applications in several domains. This research review studies the evolution of machine learning techniques applied to certain tasks in the field of parallel computing. These tasks initially required domain expertize, as the process of feature selection for the machine learning task, was done manually. However, according to recent research, it is possible to remove the requirement of domain experts, by automating the process of feature extraction through neural networks. Further, based on the nature of the problem being studied, transfer learning techniques can be employed to solve certain problems. This leads to the complete automation of the machine learning task and drastically reduces the manual effort required.

Signature: Date: Sunday 13th January, 2019

Supervisor: Timothy Hospedales

Contents

1 Introduction

Machine learning has found applications in several domains like image recognition [1], handwriting recognition [2] and so on. It has, in fact, found application in specialized domains like compilers [3, 4], where it is used to tune the compiler parameters which help improve the performance of applications run on the hardware. This research review aims at exploring the evolution of machine learning techniques applied to two particular problems in the field of parallel computing. These two specific problems have been chosen for the purpose of this research review because only recent research, [5] automates the entire process of feature extraction and prediction for these tasks. Prior to this, there has been extensive research to apply machine learning for constructing the heuristics of these problems but they all had the requirement of human intervention and strong domain knowledge. The research conducted to the point before the end-to-end automation, allow direct comparison of the new system with the previous systems created.

The first problem (Task 1) is based on choosing the right device, a CPU or GPU for an input code to run on. This is an important problem because a critical factor in speeding up an application is determining the right hardware (CPU/GPU) to run the application. GPUs allow massive levels of parallelization which a CPU cannot provide. However, not all programs are suited for GPU.

The second problem (Task 2) aims at choosing the rights numbers of GPU threads to coarse. Thread coercion is a process where two or more GPU threads are merged. This approach reduces extremely fine-grained parallelism in GPU, by reducing the number of threads created.

This research review studies the evolution of solutions for the two tasks (Task 1 and Task 2) in chronological order. First, these tasks are studied independently to understand the problems being addressed, followed by a description of transfer learning method to solve Task 2 using a solution developed for Task 1.

2 Literature Review

2.1 Task 1: Deciding the hardware to run the code - GPU or CPU

This section focuses on choosing the right hardware to run an input code. This hardware can be a CPU or GPU. When compared to CPU, a GPU allows much larger levels of parallelism. However, not every code is suitable to run on a GPU. Knowledge of the right hardware to run the code can help in obtaining considerable speedup in the application.

2.1.1 Seminal work on converting openMP code to GPU code

OpenMP is an API that enables multi-platform shared memory programming. It is a high-level language and is mainly used for shared memory across multi-core CPU systems. OpenCL is a standard for GPU systems and most of the code for GPU is written with this standard. The first known work in the field of translating openMP code to GPU code was targeted towards a more specific CUDA code in [6].

In [6] key transformations were performed, to enable openMP code to be converted to CUDA (Compute Unified Device architecture) code. During the time of this research publication, a baseline method existed that enabled the conversion of openMP code to CUDA code, but it led to poor results on the GPU. This was because the memory accesses were not coalesced.

The OpenMP code was optimized in two stages. In stage 1 the OpenMP stream optimizer transformed the OpenMP code to code suitable for the GPGPU using techniques like parallel loop-swap and loop-collapsing. In stage 2 the optimized OpenMP code was converted to CUDA code using a baseline translator and exploiting CUDA specific features.

[6] develops code transformations and heuristic like parallel loop swap, caching frequently accessed global data and so on. These heuristics were aimed at obtaining a more efficient GPU code from CPU code. OpenMP streams were used to further improve the code and make it as near to performance as that of handwritten CUDA. The code transformations and heuristics developed in [6] require a strong understanding of the underlying hardware and hence requires tremendous subject expertize.

The converted code was tested on two NAS parallel benchmarks (Ep and Gp) [11]. The research in [6] was mainly focused on converting openMP code to CUDA code, which limited the GPU platform to NVIDIA. This could be because openCL (the code standard for GPU systems was only released in 2009). Since [6], was also presented, around the same time (2009), it is possible that the researchers may have not had the time to immediately extend their work to openCL.

2.1.2 Machine Learning to predict code mapping to CPU/GPU

According to [7] the drawbacks in [6] are as follows. There is no transformation step to optimize the code for all GPU architectures. The second drawback is that the program always runs on GPU. While GPU allows massive levels of parallelization, when compared to CPU it is not superior in every case. This is explained at length in [8] which concludes that under certain scenarios there are applications that have performance that either match up to, or are even better than their performance on the GPU. [9] states that CPU and GPU have, a performance that are comparable when applied with the right code optimization methods like multi threading and cache blocking for CPU, and using local shared buffers for GPU. The optimization technique that can be applied to an input code depends on the nature of the code itself and hence a certain code can be optimized to perform better on either of the component (CPU/GPU), of the heterogeneous system.

The approach used in [7] involves two stages, the first is the compile-time stage and the second is run-time stage.

At **compile time** the openMP program is converted to OpenCL code . As a part of the code transformation step, a number of optimizations are performed on the OpenCL code to ensure that the OpenCL code obtained is efficient for GPU architectures. According to [7] these steps include optimizations of local memory, reordering memory loads, reordering global indexes and loop interchanges. These transformations help in efficient memory access. The OpenCL code is then used to extract features for the predictive model. The features used for the predictive model are shown in Table 1. These code features are selected by domain experts (in [7] it was the compiler writer that decided the features to use for the predictive model).

Feature	Description
F1	Commun.-Computation Ratio
F2	Percentage of coalesced memory accesses
F3	Average number of work items per kernel × Ratio of local to global memory accesses
F4	Memory ratio

Table 1: Features used for the predictive model in [7]

The machine learning predictor is built off-line by constructing decision trees using the C4.5

algorithm [10]. The training set consists of a set of programs that are run on GPU and CPU. The features extracted from the code serve as features for the training set and the right device to run the code on serve as the corresponding labels for the training set. Hence this methodology falls under the category of supervised learning. Training the model is a one time cost and according to [7] takes even lesser than a day to train on a machine. Several benchmarks like NAS parallel benchmark[11], AMD Accelerated Parallel Processing SDK[12], NVIDIA CUDA SDK[13] and so on were used for training the model.

At **run time** the features extracted in compile time are parameterized based on the values obtained in run time. Function pointers to both the versions of code (CPU/GPU) exist. The predictive model then predicts the right hardware to run the application. Based on the choice of the hardware the predictive model picks either the OpenMP code to run on the CPU or the OpenCL code to run on the GPU. As per [7] the overhead for this prediction is just a few microseconds.

The final outcome of [7] was that it had achieved a speedup of around 4 times when compared to the baseline. However, the fact that there are two versions of the code (one for the CPU and the other for the GPU) implies that there is a duplication of code. It would have been ideal if there was a method by which it was possible to determine the right hardware at the compile time itself and discard the less efficient code version before proceeding to the runtime. In [7] this was not possible, because of the parameterized form of the features which require values that can only be populated at run-time. The merit of the system however, stems from the fact that during run time there is very little overhead for the prediction and once the right hardware is known the speedup is around 4 times, as stated in [7].

2.1.3 Improving performance by synthesizing benchmarks

It is possible to improve the performance of the model in [7] by increasing the number of training examples for the predictive model. However, as per [14] there is a shortage of training examples in the field of machine learning for compilers and this creates a limitation for the learned models. The already existing benchmarks are not sufficient to completely cover the entire feature space. Hence a method to counter this problem is by synthesizing the benchmarks. [14] uses deep learning to synthesize programs that mimic programs written by human developers. It is important that the synthesized program is similar to the program written by human developers because the learning has to target the right parts of the features space.

The research in [14] creates an OpnCL program generator called "CLgen" that can synthesize programs which can be used to train the predictive models. Synthesizing such benchmarks exposes the fact that covering the feature space finely brings out the weaknesses in handcrafted features like the ones chosen in [7]. To be able to generate the OpenCL code, a large number of OpenCL code were first mined from public repositories on GitHub. [14] then uses LSTM (Long Short-Term Memory) architecture of Recurrent Neural Networks, which learns a language model at the character level, over the OpenCL training examples. This is an unsupervised machine learning task. A seed text is then generated and the models are sampled on a character by character basis to generate the OpenCL code. The OpenCL code is generated from the seed text according to the Algorithm 1 (Figure 1). In [14] human beings were involved, to directly verify that the generated OpenCL code was indeed human like. According to [14], this was a justified step as it draws from the application of Turing Test in machine learning research. Turing Test has previously been applied to machine learning tasks like question answering system of images [15] and image colorization [16]. Turing test developed by Alan Turing [17], employs a human evaluator to differentiate between a machine and human while having a conversation in natural

language with both. If the human evaluator is unable to differentiate between the machine and the human, then the machine is said to have passed the Turing test.

Figure 1: Algorithm 1 drawn from [14]

Algorithm 1 Sampling a candidate kernel from a seed text.

Require: LSTM model M, maximum kernel length n.
Ensure: Completed sample string S.

```
S ← "__kernel void A(const int a) {"          Seed text
d ← 1                                         Initial code block depth
for i ← |S| to n do
    c ← predictcharacter(M, S)                Generate new character
    if c = "{" then
        d ← d + 1                             Entered code block, increase depth
    else if c = "}" then
        d ← d − 1                             Exited code block, decrease depth
    end if
    S ← S + c                                 Append new character
    if depth = 0 then
        break                                 Exited function block, stop sampling
    end if
end for
```

By using a larger number of training examples it was observed in [14] that, the Feature F3 drawn from [7] (presented in Table 1 of this report) was sparse for many programs. This was due to the nature of the training set examples in the NPB training suite [11]. To counter this problem, [14] then used raw feature values (like the local emery access, global memory access and average number of work items per kernel) and the combined feature values in the training set. Another drawback in the training set of [7] was the absence of differentiating features in the case of branching code. This was again due to the nature of code in the NAS benchmark which was used for the training set in [7]. To counter this problem a new feature was added, which is a count of the branching operations in the kernel. The modification to the features helped to further improve the performance on an average by around 4× (5.04 on NVIDIA and 3.56 on AMD).

From the research in [14] it can be observed that the hand-picked features despite being chosen by domain experts (the compiler writer in [7]) had room for further improvement. This was achieved by working on the idea of expanding the dataset used for the training. However, it should be noted that human intervention was still required to tweak the features. [14] did not automate the process of feature extraction. In fact, it's purpose was to obtain a method of generating a larger dataset that can be used for the training. Weakness in the features was detected by considering the larger dataset.

2.1.4 Using Deep Learning to achieve complete automation

In [5], the training features are learned over raw code by neural networks. The neural network constructs the internal code representation and learns code optimization. This process essentially eliminates the process of manual feature selection and hence, making the process of feature selection cheaper and much faster. The advantage of learning on the source code is that, the solution is not tied to any specific platform or compiler. The process of feature learning and prediction (CPU/GPU) involves the following stages.

The target code is first fed to a code rewriter which also performs the code transformations stated in [14]. This stage removes the comments (semantically irrelevant information) from the

target program and ensures trivial code differences like variable names do not affect the learned model.

The output from the source rewriter is then fed to the language model which converts the source code into a vector of fixed length, capturing the code properties and structure. The source code is encoded as a sequence of integers using the language model developed in [5]. This language model is a hybrid of the character model in [14] and vocabulary based on tokens model in [18]. The keywords in the OpenCL language are assigned unique integers and the literals are encoded at character level. For example in [5] 'void' is encoded with the integer 3 and the literal 0 is mapped to '0' and encoded with the integer 23. The encoded source code is then mapped to vector embeddings.

Several OpenCL programs are encoded this way and along with the training labels (CPU/GPU) serve as the training set for the neural network. The neural network architecture consists of a two layer LSTM (Long Short Term memory) form of Recurrent Neural Network. The output of the 2 layer LSTM is a single vector that covers the entire code sequence. This step is the equivalent of finding features by domain experts in [7]. The output of the 2 layer LSTM is then concatenated with auxiliary inputs (dynamic values which cannot be determined from the static code features) like work group size and data size. The concatenated vector is then fed through a deep neural network connected to two outputs, one depicting the CPU and the other a GPU. The structure of the neural network is depicted in figure 3 (a) in section 2.2.3. This model is evaluated using a 10 fold cross-validation technique [19]. When compared to [7], the model developed in [5] had achieved a speedup of 3.34× in AMD and 1.41× in NVIDIA.

On comparing to the approach adopted in the previous section "Improving performance by synthesizing benchmarks". [14] had achieved a speedup of 3.56× on AMD and 5.04× on NVIDIA. This is greater than the speedup achieved in this section which is, 3.34× in AMD and 1.41× in NVIDIA. However, the salient points to consider in this case is [14] still required human effort to decide the features. It was aimed at improving the size of the training set for the CPU/GPU problem. [5] has achieved the speedup without including the expanded training set and was trained on the same OpenCL programs that [7] was trained on. The neural network approach totally eliminates the process of feature selection which requires an in-depth domain knowledge. This is one of the greatest merits of the method, devised in [5].

Future work in this specific research could be to use the expanded training set developed in [14] to train the neural network method developed in [5]. The performance of this model can then be used to compare the performance of the machine learning model in [7].

2.2 Task 2: Thread Coarsening for Graphic Processors

Thread coercion is the process of merging together two or more parallel threads in a GPU. The challenge in this task is, to choose the optimal number of threads to coerce such that the execution of redundant instructions is avoided while still leveraging the power of parallelism. The solution to this problem tackles over-parallelism at the thread level.

2.2.1 Understanding thread coercion and its effects

The research [20], studies the effects of thread coercion and mentions the importance of choosing the right number of threads to coerce. This is because as per [20] thread coercion removes instructions that are redundant. The coercing factor is not only specific to an application but also to the hardware on which it is run. This fact is supported by empirical evidence in Figure 2.

Figure 2: Image obtained from [20]

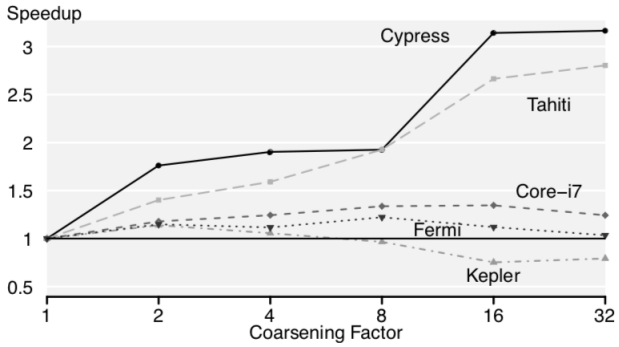


Figure 2 depicts the speedup of application (finding the transpose of a matrix) on various hardware and with varying thread coarsening factors. The Cypress and Tahiti are AMD GPUs, Fermi and Kepler are NVIDIA GPUs and Core-i7 is an Intel CPU [20]. From the figure 2 it can be deduced that while increasing the thread coercion factor has a positive effect on certain hardware like Cypress, on Kepler increasing the thread coercion beyond 2 has a negative effect and on Fermi the optimal coercion factor is 8. Hence, it can be concluded that thread coercion factor depends not only on the application itself but also on the hardware on which it is run on. To apply thread coercion, [20] first identifies divergent instructions. These instructions are those that have different behavior in different threads (processing elements) [21]. Clones of the divergent instructions are then created and thread coarsening is then applied. Using regression trees, [20] studies independently each of the hardware (Kepler, Fermi, Cypress, Tahiti, and AMD) and the effect of important parameters like the loads, branches and so on in an application, in relation to the thread coarsening factor. It was seen that for hardware like Fermi, Kepler and Tahiti the most important factor was the load operation, for Cypress it was ALU packing and for Core-I7 it was vector instructions.

It can be concluded from the study in [20] that, the optimal thread coercion factor depends on the application and hardware it is run on. By observing the results from the regression trees in [20], it can be further concluded that different parameters in an application have varying levels of importance across different hardware. These different parameters are affected differently by the thread coercion and hence this justifies the observations in Figure 2. To determine the optimal thread coercion factor, code versions with varying levels of thread coercion have to be tested to arrive at the optimal number of threads to coerce.

2.2.2 Automatic thread coercion

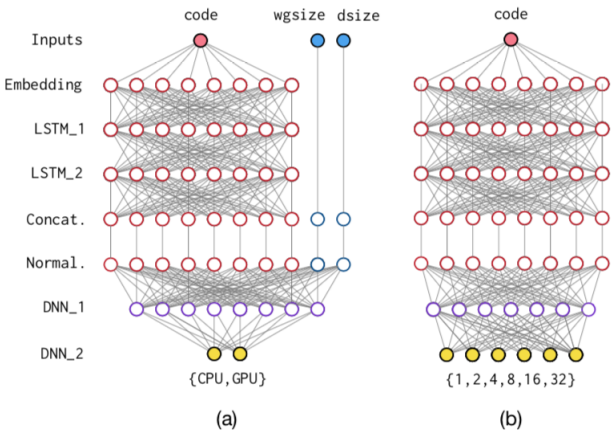
In [22], neural networks are used to determine the number of threads to coerce. The neural network is trained in 2 phases. In phase 1 of the training phase, static code features decided by the domain expert are collected from a large number of OpenCL programs. These features include the number of loads, stores, branches, divergent instructions and so on. In phase 2 of the training phase, different versions of the OpenCL programs are run with coarsening factors in the range of 2,4,8 16 and 32. A binary training output records if coarsening improved the performance. This 5 member tuple binary training output, together with the features comprise of the training set for the neural network.

To determine the coarsening factor for an unseen program, the static code features are fed to the neural network to determine if coarsening should be applied. In case the output from the neural network is yes, then the coarsened version of the program is tested on the neural network. This process is repeated iteratively until an optimal coarsening value is obtained. This can be identified by the fact that, beyond the optimal point, coarsening leads to performance degradation. As per [22], there is a performance improvement of 1.11 times on NVIDIA GPU and 1.33 times on AMD GPU.

2.2.3 Using Deep Learning and Transfer Learning to achieve complete automation

Similar to the process described in the section "Using Deep Learning to achieve complete automation" (section 2.1.4), the source code is fed to a target rewriter and the output from source rewriter is fed through the same language model developed in [5] to obtain a fixed length vector representation of the source code. The only difference, in this case, is that the training examples are different and there are no auxiliary inputs concatenated with the vector. Also, the neural network has 6 outputs (1,2,4,8,16,32) instead of 2 (CPU,GPU). The following figure (Figure 3) encapsulates these differences, wherein the figure (a) is the neural network structure for the CPU/GPU task and the figure (b) is the neural network structure for the thread coarsening task.

Figure 3: Image obtained from [5]



As per the experimental results, this setup achieved an average performance gain of 1.06×, when compared to the model developed in [22]. According to [5] the performance gain is not much because of the small training set used in [22] which consists of only 17 programs. The model built in [5] for the thread coercion problem is also trained on just these 17 programs. One way to tackle this problem is by using an expanded training set. However, [5] uses a different strategy. It uses the method of transfer learning.

According to [5] using transfer learning, it is possible to improve the accuracy of the thread coercing task. This is because in the two tasks (CPU/GPU, thread coercion) the neural network developed in [5] is used to learn the code structure. According to [5], since for both tasks the neural network has the similar input which is the OpenCL code, the weights learned in the layers near input will be useful for other tasks where the neural network has similar inputs. Hence the weights learned from the first two LSTM layers of the CPU/GPU task, were transferred to the first two LSTM layers in the neural network for the task of thread coercion. According

to [5] there was an average of 12 percent improvement in the speedup when compared to that obtained in [22]. Hence, transfer learning helps in the improvement of performance in the thread coercion task. The major advantage of this method is that there is no human effort required for the feature construction and the performance gain is obtained at little extra effort once the neural network is constructed and the weights have been learned.

3 Summary and Conclusion

For the CPU/GPU task, while [5] uses neural networks to learn the features, it still uses the limited training examples used in [7]. The experimental results prove that [5] gives an improved performance gain (speedup) when compared to [7]. However, it should be possible to obtain an even more improved performance by expanding the training set. The researchers had previously relied upon readily available benchmarks for training and testing. [14] presents a method of generating developer-like code from a machine. This can be used for expanding the training set.

Future work could be to first expand the training set and then record the performance improvement when compared to [7]. Another very important work that should be done is to create a training set of very large size using [14], that should be standardized across the scientific community and will be a superset of all the benchmarks that exist. This will genuinely help future researchers in this field, by eradicating the problem of data sparsity. It will serve as a standard benchmark on which all future models, can be built and tested on.

For the thread coercion task, a comparative study can be set up which involves a combination of expanding the training set and using transfer learning.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2018.

[2] A.W. Senior and A.J. Robinson. An off-line cursive handwriting recognition system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, 1998.

[3] V. Dalibard, M. Schaarschmidts, and E. Yoneki. Boat: Building auto-tuners with structured bayesian optimization. *Proceedings of the 26th International Conference on World Wide Web*, ACM, 2017.

[4] A.H. Ashouri, W. Killian, G. Palmero, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys*, ACM, 2018.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2017.

[6] S. Lee, S. Min, and R. Eigenmann. Openmp to gpugpu: A compiler framework for automatic translation and optimizarion. *PPoPP '09 Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[7] D. Grewe, Z. Wang, and M.F.P. O' Boyle. Portable mapping of data parallel programs to opencl for heterogenous systems. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[8] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or not! multi-core cpus can match gpu performance for a flop-intensive application! *Published in: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[9] V.W. Lee, C.Kim, J.Chhugani, M.Deisher, D.Kim, A.D. Nguyen, N.Satish, M.Smelyanskiy, S.Chennupaty, P.Hammarlund, R.Singhal, and P.Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ISCA '10 Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[10] J. R. Quinlan. C4.5: programs for machine learning. *Morgan Kaufmann Publishers*, 1993.

[11] NAS parallel benchmarks 2.3 OpenMPC version. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[12] AMD. AMD/ATI Stream SDK. `http://www.amd.com/stream/`.

[13] NVIDIA Corp NVIDIA CUDAK. `http://developer.nvidia.com/object/cuda.html`.

[14] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. *CGO '17 Proceedings of the 2017 International Symposium on Code Generation and Optimization*, 2017.

[15] H. Gao, J. Mao, J. Zhou, Z. Huang, L. Wang, and W. Xu. Are you talking to a machine? dataset and methods for multilingual image question answering. *In: arXiv:1505.05612*, 2015.

[16] R. Zhang, P. Isola, and A.A. Efros. Colorful image colorization. *In: arXiv:1603.08511*, 2016.

[17] A. M. Turing. Computing machinery and intelligence. 1950.

[18] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[19] M. Kamber, J. Pei, and J. Han. Data mining: concepts and techniques. *Elsevier*, 2011.

[20] A. Magni, C. Dubach, and M.F.P. O' Boyle. A large-scale cross architecture evaluation of thread coarsening. *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[21] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. Divergence analysis and optimizations. *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, IEEE, 2011.

[22] A. Magni, C. Dubach, and M.F.P. O' Boyle. Automatic optimization of thread-coarsening for graphic processors. *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT))*, 2014.