A

Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-based Heterogeneous Systems

Zheng Wang, Lancaster University
Dominik Grewe, University of Edinburgh
Michael F.P. O'Boyle, University of Edinburgh

General purpose GPU based systems are highly attractive as they give potentially massive performance at little cost. Realizing such potential is challenging due to the complexity of programming. This article presents a compiler based approach to automatically generate optimized OpenCL code from data-parallel OpenMP programs for GPUs. A key feature of our scheme is that it leverages existing transformations, especially data transformations, to improve performance on GPU architectures and uses automatic machine learning to build a predictive model to determine if it is worthwhile running the OpenCL code on the GPU or OpenMP code on the multi-core host. We applied our approach to the entire NAS parallel benchmark suite and evaluated it on distinct GPU based systems. We achieved average (up to) speedups of 4.51x and 4.20x (143x and 67x) on a Core i7/NVIDIA GeForce GTX580 and a Core i7/AMD Radeon 7970 platforms, respectively over a sequential baseline. Our approach achieves, on average, over 10x speedups over two state-of-the-art automatic GPU code generators.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: GPU, OpenCL, Machine-Learning Mapping

ACM Reference Format:
Zheng Wang, Dominik Grewe, and Michael F.P. O'Boyle, 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-based Heterogeneous Systems. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Heterogeneous systems consisting of a host multi-core and general purposed GPU are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. Users typically have to identify potential sections of their code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address

---

Extension of Conference Paper: a preliminary version of this article entitled "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems" by D. Grewe, Z. Wang and M. O'Boyle appeared in International Symposium on Code Generation and Optimization (CGO), 2013 [Grewe, Wang, and O'Boyle Grewe et al.].

Author's addresses: Z. Wang, School of Computing and Communications, Lancaster University; D. Grewe (current address), Google London, London; M. O'Boyle, School of Informatics, University of Edinburgh; email:z.wang@lancaster.ac.uk.

DOI: http://dx.doi.org/10.1145/0000000.0000000

space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous computing. It provides portability, allowing the same code to be executed across a variety of processors including multi-core CPUs and GPUs. By exposing fine-grained parallelism, carefully written OpenCL (or CUDA) programs can achieve good performance across parallel processor architectures [Stratton et al. 2010].

However, there are many legacy programs written with shared memory programming languages such as OpenMP [Lee et al. 2009]. To benefit from heterogeneous performance, it will require considerable development efforts to rewrite those programs with OpenCL and the process can be error-prone [Lee and Eigenmann 2010]. This work aims to provide a simple upgrade path for targeting OpenMP programs on heterogeneous platforms. We achieve this by developing a compiler based approach that automatically generates optimized OpenCL from a subset of OpenMP, and using *machine learning* to determine the best performing processor on a CPU-GPU mixed system. This allows the user to run the same data parallel program written in OpenMP, which has been fully tested, with no modifications, while benefiting automatically from heterogeneous performance.

The first effort in this direction is [Lee et al. 2009]. Here, the OpenMPC compiler generates CUDA code from OpenMP programs. While promising, there are two significant shortcomings with this approach. Firstly, OpenMPC does not apply data transformations. As shown in this article data transformation is crucial to achieve good performance on GPUs (Sections 4 and 7.4). Secondly, the programs are always executed on GPUs. While GPUs may deliver improved performance, they are not always superior to CPUs [Bordawekar et al. 2010; Lee et al. 2010a]. A technique for determining when GPU execution is beneficial is needed. This article addresses both of these issues and when evaluated on the full NAS parallel benchmarks our technique outperforms OpenMPC by a factor of 10.

Our work examines performance portability across heterogeneous platforms, considering the trade-offs in heterogeneity – it may be better to run the program as OpenMP on a multi-core rather than as OpenCL on the GPU. It uses OpenCL as a target language and is applied to the entire NAS parallel benchmark suite on different GPU based systems from different vendors, using automatic machine learning techniques to predict the best components to use in a heterogeneous system. We generate high quality OpenCL code achieving up to 202x speedup over sequential C for the `ep` benchmark on AMD Radeon automatically. This article's technical contributions can be summarized as follows. It is the first to:

— use machine learning to decide between different implementation languages on heterogeneous platforms (Section 5)
— use machine learning to automatically build cost-based models for dynamic array index re-ordering for GPUs (Section 4)
— automatically translate and map all NAS parallel benchmarks onto GPUs, some of benchmarks are up to 3600 lines long with 66 kernels – a non-trivial task (Section 7).

A key feature of our scheme is that it uses machine learning to build predictive models to automatically determine if it is worthwhile running the code on the GPU or the multi-core host. Furthermore, it can adapt this model to different GPU architectures and generations. We also show that data transformations can be used to significantly improve performance on GPU architectures. This means that the user can use the same OpenMP code on different platforms with the compiler determining the best place for code to run and optimize it accordingly.

2. MOTIVATION

With the massive interests in GPUs, it is important to know that GPUs are not always the most suitable device for scientific kernels. This section provides a simple example demonstrating the appropriateness of GPU computation depends on the original program, data size and the transformations available.

Consider the OpenMP fragment in Figure 1a from the NAS parallel benchmark `bt`, a benchmark containing over 60 parallel loops potentially suitable for offloading to a GPU. This program was executed with two different programs input: a small input, W, and a large input, A. Using our basic OpenMP to OpenCL translator yields the code shown in Figure 1b. The parallel loop has been translated into a kernel where each of the loops is parallelized forming a 3D parallel work-item space (i.e. ND-Range) each point of which is accessed through a call to `get_global_id` for dimensions 0, 1 and 2.

This code if executed on a GPU with the W input size however gives disappointing performance when compared to executing the code on a multi-core as shown in Figure 1c. This is largely due to a relatively small work to be performed on the GPU where a high percentage of coalesced memory access is required to achieve good performance (see Table IV in Section 7.6). If we execute the same code with a larger input size A, the GPU performance improves but is still less than the performance achieved on the multi-core. The main reason is the memory access pattern of the kernel which does not allow for memory coalescing on the GPU. This can be changed by performing global index reordering (see Section 4.1) as shown in Figure 1d, transforming the data layout of array `lhs`. This gives the new OpenCL program shown in Figure 1e. Here the most rapidly varying indexes of the array correspond to the tile IDs giving coalesced memory accesses. As can be seen later from Table IV, this improves the percentage of coalesce memory access (*feature F2* in Table IV) from 0 to 0.78 and 0.999 for input sizes W and A, respectively. In Figure 1f we see that the resulting performance of the GPU code improves substantially for data size W. If this transformed code is executed with a larger data size A, the GPU performance further improves, with both GPUs now outperforming the multi-core OpenMP implementation.

This example shows that the translated OpenCL code can give better performance than the original OpenMP code depending on the data size and transformations available. As described in Section 7, this decision varies from program to program and across different platforms and data sizes, and depend on a number of factors (see Figures 13 and 14). What we would like is a system that learns when to use the GPU, changing its decision based on the availability of underlying optimizations such as data layout transformations.

3. OVERALL SCHEME

Our compiler automatically translates OpenMP programs to OpenCL-based code, performing loop and array layout optimizations along the way. It generates multi-versioned code for each parallel loop: the original OpenMP parallel loop and an optimized OpenCL kernel alternative. At runtime, a machine learning (`ML`) based predictive model decides which version to use for execution. Our prototype compiler is implemented using Clang (v3.0) and LLVM (v3.0) [LLVM 2013].

3.1. Compile-Time

Figure 2 gives an overview of our approach. The OpenMP program is read in and parallel loops are optimized and translated to OpenCL kernels. The generated kernels are passed to a feature extraction phase which collects characteristics or features from the Abstract Syntax Tree of the generated OpenCL code. These features are later used by the `ML` model to select whether the OpenMP loop or OpenCL kernel version is best (Sec-

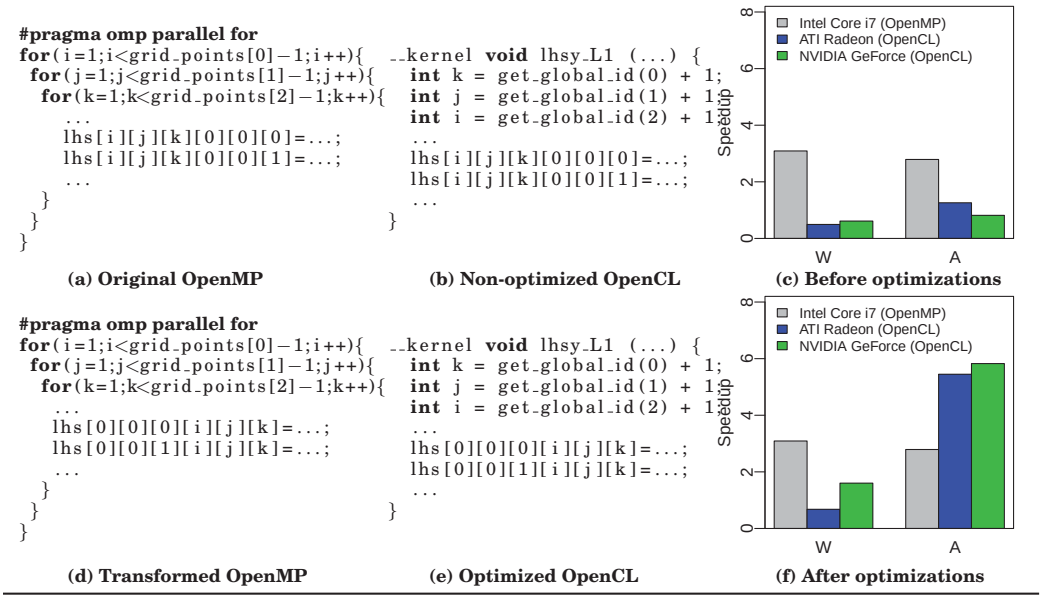


Fig. 1: Simplified example of generating OpenCL code from OpenMP code. The top left code (a) snippet is taken from `bt`. The corresponding OpenCL code (b) delivers poor performance on both GPUs (c). After applying data transformation to the OpenMP code (d), we obtain the new OpenCL code shown in (e). The performance of both GPUs improves significantly, but only for large inputs can they outperform the CPU (f).

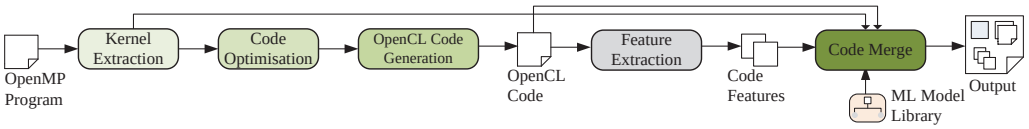


Fig. 2: Overview of our compilation framework.

tion 4). The features, together with the generated OpenCL code, the original OpenMP code and a `ML` predictor built off-line (Section 5) are merged into a single output program source.

3.2. Run-Time

At execution time, the program first updates the parameterized feature values based on the runtime values of parameters and passes the instantiated feature values to the `ML` model. The built-in model then predicts where to run the program and to pick either the OpenMP version for a multi-core CPU or the OpenCL version for a GPU. Evaluating the model at runtime involves on the order of tens of operations and is thus negligible. This is the high-level overview of the compilation framework. The next sections describe each of the stages in more detail.

4. CODE GENERATION AND OPTIMIZATION

Table I lists the OpenMP directives supported by our implementation.

Parallel Constructs etc:	`parallel`	`for`	`for reduction`	
Data Attributes:	`default` `last_private`	`shared` `threadprivate`	`private` `copyin`	`first_private`
Other Constructs:	`barrier` `single`	`atomic` `flush`	`critical`	`master`

Table I: Supported OpenMP directives.

Work-sharing Constructs. Our compiler converts OpenMP parallel loops, i.e. loops that are annotated with `omp for` or `omp for reduction`, into OpenCL kernels. Other parallel OpenMP directives associated with task parallelism are not currently supported. A two-stage algorithm [AMD 2013] is used to translate parallel reduction loops.

Data Attributes. In OpenMP variables may have additional type information specified by directives: `default`, `shared`, `private`, `first_private`, `last_private`, `copyin` and `threadprivate`. Our framework uses these directives to map data onto the GPU memory space. Each variable with the `share` or `default` directive will be translated into an OpenCL global variable shared by all OpenCL threads. Variables declared as `private` and `threadprivate` are translated such that there is a private copy for each OpenCL work item; no memory transfer between the GPU and the CPU is needed. For each variable specified as `copyin` or `first_private`, we create a private copy for each OpenCL work-item but initialize each copy using explicit memory transfers before its first use. Similarly, we create a private copy of a `last_private` variable and the original variable is updated by the GPU thread that executes the last work item.

Other Constructs. Our implementation also supports a number of synchronization and thread constructs. Structured blocks identified with `master`, `single` and `critical` directives are executed by one thread on the host multi-core. `barrier` is implemented by calling the OpenCL `clFinish` API to synchronize all OpenCL threads. An `atomic` operation is translated into the corresponding OpenCL atomic function, according to the type of the operand (variable). The current level of atomic support can be easily extended to support part of the C11 atomics, such as atomic load and stores. To fully support the C11 atomic standard is our future work. Finally, `flush` is implemented using the OpenCL `barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)` API.

4.1. OpenCL Code Optimization

Our compiler performs a number of optimizations to improve the performance of the OpenCL code on the GPU[1]. The optimizations implemented in our compiler are applied in the following order:

Loop Interchange. High memory bandwidth on GPUs can only be achieved when memory accesses are coalesced, i.e. adjacent threads access adjacent memory locations in the GPU off-chip memory. Our framework applies loop interchange to place outermost those iterators that occur most frequently in the innermost array subscripts. We use the LLVM `DependenceAnalysis` pass to detect to which level the nested loop can be interchanged.

Global Index Reordering. Global index reordering is the data structure equivalent of loop reordering. Indexes of an array are permuted to fit an optimization purpose. This transformation is necessary when loop interchange cannot provide memory coalescing. This can be represented as $[s_1, s_2, \ldots, s_n] \mapsto [s_{x_1}, s_{x_2}, \ldots, s_{x_n}]$ where each $x_k \in 1, \ldots, n \wedge \forall i, j, (x_i = x_j) \Rightarrow (i = j)$. In our case we wish to place outermost

[1]The LLVM "-O3" optimization stack also provides other types of optimizations that primarily performs on the LLVM IR level. Integrating this into our source to source OpenCL code generator is the future work.

those indexes which appear as OpenCL work-item indexes. Given that there are a maximum of three parallel work-item indexes, t_1, t_2, t_3, then for any one access we wish $s_{x_n} = t_1, s_{x_{n-1}} = t_2, s_{x_{n-2}} = t_3$. An example of this transformation was shown in Figure 1: $[i, j, k, 0, 0, 0] \mapsto [0, 0, 0, i, j, k]$.

After index re-ordering, our compiler renames the function prototype where input array arguments will be updated accordingly, and changes the code where this function is called. This removes the need for additional array copying (in and out) in order to comply with the calling context. Global index reordering requires global alias analysis to ensure correctness. To do so, we run the code through the "-basicaa" and "-steens-aa" passes provided by LLVM, where the later pass implements the well-known Steensgaard's pointer analysis [Steensgaard 1996]. Note that our current implementation only applies global index reordering to statically allocated arrays.

Dynamic Index Reordering. In conjunction with loop interchange global index reordering is often sufficient to achieve memory coalescing. However, in some cases there is no clear best global data layout, e.g. when different loops in a program require different layouts to achieve memory coalescing. We then consider *dynamic* index reordering [Che et al. 2011].

Before entering a code region containing loops that prefer a certain index order for an array X (different from the global one), a reordered copy of the array, X', is created. During the copy process the data gets reordered as follows. Given a preferred access pattern $[s_{x_1}, s_{x_2}, \ldots, s_{x_n}]$ we copy X to X' such that $X'[s_{x_1}, s_{x_2}, \ldots, s_{x_n}] = X[s_1, s_2, \ldots, s_n]$. Within the code section all references to X are redirected to X' and the indexes are reordered appropriately. At the end of the region the data gets copied back to the original array.

Dynamic index reordering for GPU computing can often be prohibitive. The transformation should only be applied if the benefits of data coalescing outweigh the costs of data reordering. To be used in a compiler setting we therefore need a mechanism to automatically determine when this transformation should be applied. Section 5.4 describes a `ML` based cost model that solves this problem.

Memory Load Reordering. In the original OpenMP programs, accesses to read-only buffers might be reordered to form a sequence of consecutive load operations that can be vectorized. Our compiler automatically detects those accesses and replaces scalar load operations with an OpenCL vector load operation. Our current implementation simply groups consecutive load and store operations together and replace them with an OpenCL `vloadn` or `vstoren` ($n = 2, 4, 8, 16$) operation. This is a standard optimization technique that has been used in prior work [Eichenberger et al. 2004; Yang et al. 2010].

Register Promotion. On many occasions, a scalar variable (or array) that stored in the global memory space is accessed multiple times by a single OpenCL kernel. To reduce global memory latencies, our tool automatically creates a private register object for such a variable. It generates code to load the data from the global memory to the private register copy (and write back to the global memory from the register after the last store operation). Doing so can eliminate redundant global memory loads and stores and thus improve performance. Depending on the implementation, the back-end OpenCL compiler might also perform register promotion optimization [Lu and Cooper 1997; Chakrabarti et al. 2012] on the generated code.

Prefetching and Local Memory Optimization. For read-only buffers (that are used by multiple GPU threads) identified by our compiler, we generate code to prefetch the data to the local memory. Exploiting local memory in general reduces memory latencies for GPU kernels [Lee et al. 2010b].

```
1  static void __ocl_lhsy_L1(...) {
     size_t __ocl_gws[3] = {((grid_points[2]-1)-(1)),...};
3    oclSetKernelBuffer(__ocl_lhsy_L1,3,__ocl_buffer_lhs);
     ...
5    oclWrites(__gpu,___ocl_buffer_lhs);
     ...
7    oclRunKernel(__ocl_compute_rhs_3,3,__ocl_gws);
   }
9
   static void lhsy() {
11   if (unlikely(!__ocl_has_pred)) {
       __predict();
13   }
     __p_lhsy_L1(...);
15   ...
   }
17
   int main() {
19   lhsy();
     ...
21   oclReads(__host,__ocl_buffer_lhs);
     oclSync();
23   ...
   }
```

Fig. 3: The generated host code of the loop shown in figure 1(a).

Host-Device Communication. For each array that is used by both the host and the GPU we manage two copies: one on the host memory and the other on the GPU memory. Our runtime records the status of each variable and checks whether the copy on a device memory space is valid or not. No memory transfer is needed as long as the copy in the target memory space is valid. Our current implementation uses a conservative approach: if an element of an array has been updated, the entire array needs to be synchronised before it can be used by threads running on a different device. There are advanced techniques available for host-device communication optimizations [Jablin et al. 2012; Margiolas and O'Boyle 2014], which are orthogonal to our approach.

4.2. Host Code Generation

For each parallel loop, we outline the loop body and generate two versions for it: an OpenCL and an OpenMP version. The original loop body is replaced with a function pointer which points to either the OpenCL or OpenMP version of the loop. For example, the OpenCL version of the original loop shown in Figure 1 (a) is translated to `__ocl_lhsy_L1` as shown in Figure 3. We try to generate as many work items as possible to utilize the GPU. Each iteration of the nested parallel loops is translated into a OpenCL work item. The dimension of the ND-range is determined by the number of nested loop to be parallelized (up to 3 dimensions due to the restriction of OpenCL). In Figure 3 the OpenCL work item indexes are calculated at Line 2, which are then sent to the OpenCL runtime at line 7. The original loop body is replaced with a function pointer at Line 14. Each generated code has a prediction function, `__predict`, that decides which device to use to run the program. This is done by setting the function pointer of each loop to the corresponding code version. Currently we use a single program version for all parallel loops and do not use both versions interleaved at runtime. Finally, host-device communication is managed by our communication library where read and write operations to OpenCL buffers are identified through the `oclReads` and `oclWirtes` functions respectively (Line 5). The `oclWirtes` function keep a track of which device has recently updated the input OpenCL buffer. This information will be used to determine whether a data transfer is needed at runtime.

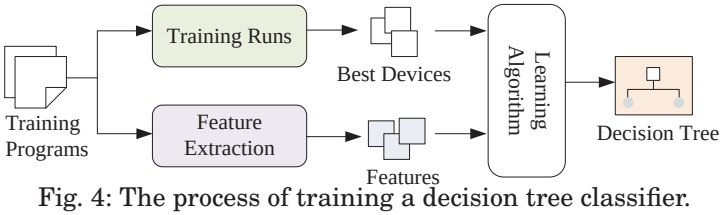


Fig. 4: The process of training a decision tree classifier.

5. PREDICTING THE MAPPING

A crucial part of our approach is to automatically determine the best computing device for the input program i.e. should it be run on the multi-core host or translated into OpenCL and executed on the GPU. Our approach is to generate the OpenCL-based code and then use a `ML` model to see if this is profitable to run on a GPU. If it is not profitable, we fall back to the original OpenMP code. As this decision will vary greatly depending on GPU architectures and the maturity of the OpenCL runtime, we wish to build a portable model that can adapt to the change of the architecture and runtime.

We wish to avoid any additional profiling runs or exhaustive search over different data sets, so our decision is based on static compiler analysis of the abstract syntax tree and runtime parameters. The static analysis characterizes a kernel as a fixed vector of real values, or *features*.

5.1. Training the Predictor

Figure 4 shows the process of training the predictor – in our case a decision tree classifier. This involves the collection of a set of training data which is used to fit the model to the problem at hand. In this work, we use a set of programs that are each executed on the CPU and the GPU to determine the best device in each case. We also extract code features for each program as described in the following section. The features together with the best device for each program from the training data are used to build the model. Since training is only performed once at the factory, it is a one-off cost. In our case the overall training process takes less than a day on a single machine.

5.2. Code Features

Our predictor is based on code features (Table IIa). These are selected by the compiler writer and summarize what are thought to be significant costs in determining the mapping. At compile time we analyze the OpenCL code and extract information about the number and type of operations. We developed a Clang-based tool to extract those features from the abstract syntax tree of the code. Double precision floating point operations are given a higher weight (4x) than single precision operations as they are expensive on many GPU architectures. Obviously, this weight is highly platform dependent, but this can be estimated by using micro-benchmarks to test how much extract time is needed for performing the same number of double floating point operations vs single floating operations. Using a analytical model similar to [Sim et al. 2012], we also analyze the memory access pattern to determine whether an access to global memory is coalesced or not. A potential feature is the amount of control flow in an application. While this feature can have an impact on performance on the GPU it was not relevant for the benchmarks we considered. It is thus not included in our feature set. This may be needed for other OpenCL application domains and it can be easily integrated into our model. Finally, instead of using raw features, we group several features to form combined normalized features that carry more information than their parts (Table IIb).

Raw Code Features	
`comp`	# compute operations
`mem`	# accesses to global memory
`localmem`	# accesses to local memory
`coalesced`	# coalesced memory accesses
`transfer`	amount of data transfers
`avgws`	average # work-items per kernel

(a) Individual code features

Combined Code Features	
F1: `transfer/(comp+mem)`	commun.-computation ratio
F2: `coalesced/mem`	% coalesced memory accesses
F3: `(localmem/mem)` $\times$ `avgws`	ratio local to global mem accesses $\times$ avg. # work-items per kernel
F4: `comp/mem`	computation-mem ratio

(b) Combinations of raw features

Table II: List of code features.

Collecting Training Data. We use two sets of benchmarks to train our model. First we use a collection of 47 OpenCL kernels taken from various sources: SHOC [Danalis et al. 2010], Parboil [UIUC 2013], NVIDIA CUDA SDK [NVIDIA Corp. 2013] and AMD Accelerated Parallel Processing SDK [AMD 2013]. These benchmarks are mostly single precision with only one kernel in each program whereas the NAS benchmarks are double precision and have multiple kernels. We thus also add the NAS benchmarks to our training set, but exclude the one that we make a prediction for (see Section 6.2).

Predictive Modeling. Our decision tree based model is constructed using the C4.5 algorithm [Quinlan 1993]. The model is automatically built from training data by correlating features to the best performing device (Figure 4). The model performs only on the generated OpenCL code. As such, it is independent on the type of the input program of our compiler. An example of a decision tree is given in Figure 13 where a decision is made by comparing one of the combined features (Table IIb) to a threshold. If the feature is smaller than the threshold the left subtree is traversed, otherwise the right one is traversed. This is repeated until a leaf node is reached which is labeled with one of the classes – "CPU" or "GPU" – telling us which device to use.

5.3. Runtime Deployment

Once we have built the `ML` model as described above, we can insert the model together with the syntax code features (extracted at compile time) to the generated code for any *unseen, new* programs, so that the model can be used at runtime.

Updating Feature Values. At compile time the OpenCL kernel is analyzed and code features are extracted and inserted to the generated program together with the trained `ML` model. As some loop bounds are dependent on the input, the compiler might be unable to determine certain feature values. These features are represented as static symbolic pre-computation of loop bound variables, which will be updated using runtime values at runtime. If the loop bounds still cannot be determined at the time the prediction function is called, we simply use the average loop bound value as an estimation.

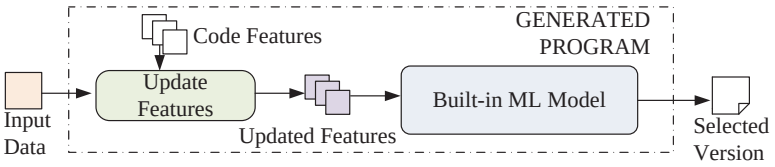


Fig. 5: Run time: selecting a code version at execution time. The generated program first updates features based on input data and passes the updated features to the `ML` model. The model then predicts the best device (CPU or GPU) to run the program and selects a code version (OpenMP or OpenCL) for execution.

Version Selection. Figure 5 depicts the process of version selection during runtime. The first time a kernel is called the built-in predictive model selects a code version for execution (Line 12 in Figure 3). It uses instantiated feature values to predict the best computing device to use and sets function pointers to the corresponding code version. In our current implementation, prediction happens once during a program's execution. The overhead of prediction is negligible (a few microseconds). This cost is included in our later results.

5.4. A Model for Dynamic Index Reordering

In Section 4.1 we described the dynamic index reordering transformation. This transformation can greatly improve performance on the GPU but it can also lead to slowdowns if the cost of reordering the data is higher than the benefits. Because the point at which the benefits outweigh the costs is highly machine-dependent we are using a portable machine learning approach that can be easily adapted to different systems. Similar to predicting the mapping we use a decision tree classifier. The features are the size of the data structure and the ratio between the number of accesses to the data structure and its size.

We developed a set of micro benchmarks and use them to obtain the training data for this problem[2]. We measure the execution time with and without applying dynamic index reordering to determine whether it is beneficial in each case. Evaluating the benchmarks and then building the decision tree model takes less than half an hour.

The resulting model is embedded into each output program because array dimensions and loop bounds may not be known at compile time. We thus keep two versions of each candidate kernel: the original one and one with accesses re-ordered. At runtime one of them gets picked by the model.

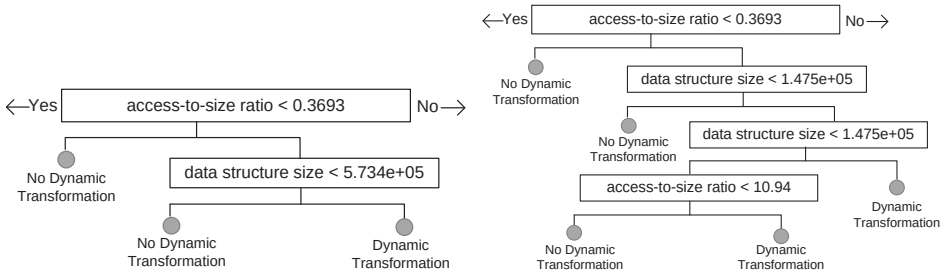
Model. Figure 6 shows the decision trees for dynamic index reordering transformation on the two GPU platforms. We used the C4.5 algorithm [Quinlan 1993] to construct the decision tree model The model is automatically built from training data which consists of data points describing a particular scenario for dynamic index reordering and whether it is beneficial or not. Each scenario is represented by two features: the size of the data structure and the ratio between the number of accesses and the size (corresponding to benefits over costs).

6. EXPERIMENTAL METHODOLOGY

6.1. Platforms and Benchmarks

Our main experiments were performed on two CPU-GPU systems: both use an Intel Core i7 6-core CPU. One system contains an NVIDIA GeForce GTX 580 GPU, the sec-

[2]We opted for micro benchmarks because the amount of training data from real applications is limited.



(a) NVIDIA GeForce GTX580 (b) AMD Radeon HD7970

Fig. 6: The automatically constructed decision trees for dynamic index reordering transformation on NVIDIA GeForce and AMD Radeon GPU platforms.

	Intel CPU	**NVIDIA GPU**	**AMD GPU**
Model	Core i7 3820	GeForce GTX 580	Radeon 7970
Core Clock	3.6 GHz	1544 MHz	925 MHz
Core Count	4 (8 w/ HT)	512	2048
Memory	12 GB	1.5 GB	3 GB
Peak Performance	122 GFLOPS	1581 GFLOPS	3789 GFLOPS

Table III: The primary evaluation hardware platforms

ond an AMD Radeon 7970. Both run with the Ubuntu 10.10 64-bit OS. Table III gives detailed information on our platforms. We also evaluated our approaches on other GPU platforms which are described in the sections where the results are presented.

All eight of the NAS parallel benchmarks (v2.3) were used for evaluation. We used the OpenMP C translation of the NAS 2.3 benchmark suite derived from the Omni compiler project [The Omini Compiler Project 2009]. Unlike many GPU benchmarks that are *single* precision, all the benchmarks except `is` are *double* precision programs.

6.2. Methodology

We considered all input sizes (S, W, A, B, C) for each NAS benchmark as long as the required memory fits into the GPU memory. All programs have been compiled using GCC 4.4.1 with the "-O3" option. Each experiment was repeated 5 times and the average execution time was recorded. The variation of runtime is small, less than 5%.

We use *leave-one-out cross-validation* to train and evaluate our `ML` model for predicting the best computing device. This means we remove the target program to be predicted from the training program set and then build a model based on the *remaining* programs. We repeat this procedure for each NAS benchmark in turn. It is a standard evaluation methodology, providing an estimate of the generalization ability of a `ML` model in predicting an *unseen* program. This approach is not necessary for the dynamic index reordering model because we use micro benchmarks as training data rather than the programs themselves.

Since OpenCL programs can also run on the CPU, an interesting question is to consider running the OpenCL programs on the CPU. However, for the benchmarks we used, doing so (with CPU-specific optimization) does not give advantages over other schemes considered in this article. On only one occasion, `bt.W`, such a scheme gives slightly better performance (8% faster) than other schemes. We also observed this on the CPU-tuned SNU OpenCL implementation. This may change when using other sets

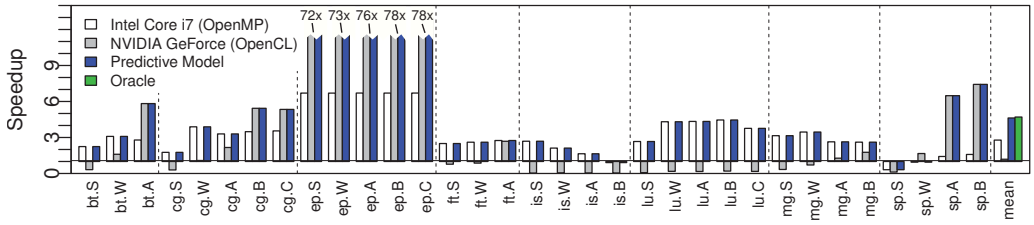


Fig. 7: Performance of OpenMP on Intel CPU, OpenCL on NVIDIA GeForce GTX580 GPU and the version selected by our predictive model. The predictive model outperforms the CPU-only approach by 1.69x and the GPU-only approach by 3.9x.

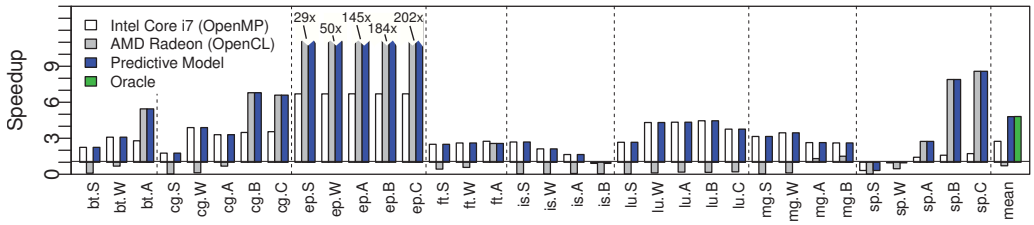


Fig. 8: Performance of OpenMP on Intel CPU, OpenCL on AMD Radeon HD7970 GPU and the version selected by our predictive model. The predictive model outperforms the CPU-only approach by 1.76x and the GPU-only approach by 6.8x.

of benchmarks, hardware platforms or OpenCL runtime, and is an interesting future research direction.

7. EXPERIMENTAL RESULTS

In this section we evaluate our approach on several heterogeneous systems for the NAS parallel benchmark suite. We first show the performance of our predictive modeling approach compared to using always the multi-core CPU or always the GPU. This is followed by a comparison to two state-of-the-art GPU code generation approaches: OpenMPC [Lee et al. 2009] and OpenACC [The Portland Group 2010]. We then compare our approach against a manual OpenCL implementation of the NAS benchmark suite [Seo et al. 2011]. Next, we provide detailed analysis of our approach, including the performance break down of different optimization strategies, OpenCL runtime and a close look of our predictive models. Finally we present a brief evaluation of our approach on two heterogeneous systems with integrated GPUs.

7.1. Overall Performance

Figures 7 and 8 show speedups for the NAS benchmarks on the two heterogeneous systems described in section 6. For each benchmark-input pair the multi-core CPU performance, the GPU performance and the performance of the device selected by our predictor is shown. The last column represents the average performance (using the geometric mean) of each approach as well as of the “oracle” which always picks the best device in each case. The performance numbers presented are speedups over single-core execution.

On both systems significant speedups can be achieved by selecting the right device, CPU or GPU. When always selecting the faster of the two speedups of 4.70x on the

NVIDIA system and 4.81x on the AMD system can be achieved. This compares to 2.78x and 2.74x when always using the CPU[3] and 1.19x and 0.71x on the GPU.

The results show that speedups vary dramatically between the CPU and GPU and none of the devices consistently outperforms the other. On `ep`, for example, an embarrasingly parallel benchmark, the GPU clearly outperforms the multi-core CPU: up to 11.6x on NVIDIA and 30.2x on AMD. However, on other benchmarks, such as `is` or `lu` the CPU is significantly faster. In the case of `lu` this is because the OpenMP version exploits pipeline parallelism using a combination of asynchronous parallel loops and a bit-array to coordinate pipeline stages. The current SIMD-like execution models of GPUs are not designed to exploit this type of parallelism. The `is` benchmark does not perform significant amount of computation and GPU execution is dominated by communication with the host memory. This leads to underutilization of the GPU and thus bad performance.

For benchmarks `bt`, `cg` and `sp` we observe that the CPU is faster for small inputs but the GPU is better on larger input sizes. This behavior is to be expected because GPUs require large amounts of computation to fully exploit their resources. On small inputs the overheads of communication with the host dominate the overall runtime when using the GPU. A similar pattern is shown for `ft` and `mg`: GPU performance is stronger for larger inputs. However, the GPU is not able to beat the CPU even for the largest input sets. For *is*, because the program does not have enough parallelism, it is actually not worthwhile to run it in parallel for any given data set on our platforms. This is also reported in other studies [Tournavitis et al. 2009].

These observations show the need for a careful mapping of applications to devices. Our model for predicting the mapping is able to choose the correct device almost all of the time. On the NVIDIA system it incorrectly picks the GPU for benchmark `sp.W` and on the AMD system it picks the GPU for `ft.A` even though the CPU is faster. Overall we are able to achieve speedups of 4.63x and 4.80x respectively. This is significantly better than always choosing the same device and not far off the performance of the "oracle".

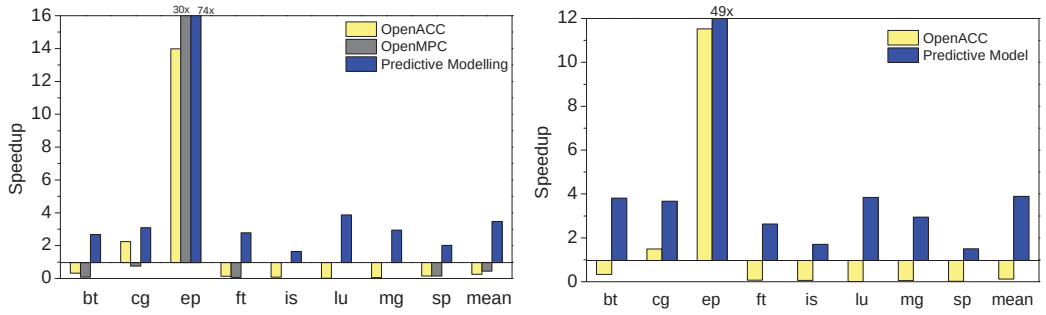
As can be seen, the best performance depends on the platform, transformations available and data sizes. Our scheme is able to predict the right option, achieving 95% of oracle performance on the NVIDIA GTX580 system and even 99% on the AMD HD7970 system.

Prediction accuracy. Our predictive model picks the correct device in 32 of the 33 cases (except for `sp`.W) on the NVIDIA system (97% accuracy) and in 33 of the 34 cases (except for `ft`.A) on the AMD system (97% accuracy).

7.2. Comparison to State-of-the-Art GPU Code Generators

We compared our approach to two automatic GPU code generation systems: (1) OpenMPC [Lee et al. 2009] which translates OpenMP to CUDA; and (2) the PGI OpenACC compiler (v 14.4) with the '-fast' and accelerator-specific optimization flags [OpenACC 2013; Wolfe 2010; The Portland Group 2010]. Because the PGI compiler fails to directly compile the OpenMP version of the NAS benchmark suite, we used a GPU-specific OpenACC implementation of the same benchmark suite developed by independent developers [PathScale Inc 2013]. Because OpenMPC generates CUDA instead of OpenCL code, we only evaluated it on the NVIDIA platform. Note that we were unable to generate code for benchmarks `is`, `lu` and `mg` using OpenMPC. The results are shown in Figure 9.

[3]Even though the same CPU was used in both cases the numbers vary slightly because the benchmarks sets are different due to memory constraints on the GPUs.



(a) NVIDIA GeForce GTX580 (b) AMD Radeon HD7970

Fig. 9: Speedup averaged across inputs of the OpenMPC compiler (NVIDIA only), the manual SNU implementation of the NAS benchmark suite and our predictive model.

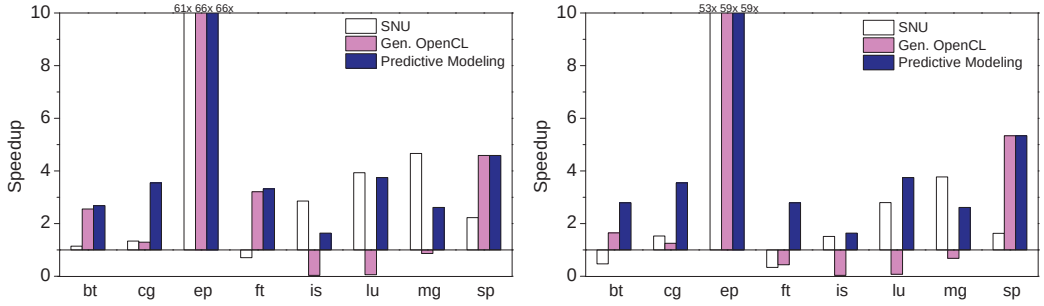
With the exception of `ep`, the OpenMPC generated code performs poorly compared to our approach. It only achieves a mean speedup of 0.42x, i.e. slower than sequential execution. The main reason is that OpenMPC does not perform data transformation, leading to uncoalesced memory accesses in many cases. On average, our approach outperforms OpenMPC by a factor of 10.

The PGI OpenACC compiler also gives overall slowdown performance. Note that by excluding `is`, `lu`, and `mg` which OpenMPC fails to compile, the PGI OpenACC compiler actually gives better overall performance than OpenMPC (0.7 vs 0.4). OpenACC outperforms OpenMPC on the `cg` benchmark but delivers poorer performance than our approach on both platform. On average it gives slowdown performance instead of a 4.18x and 3.9x speedup achieved by our approach on the NVIDIA and AMD platform respectively. Because the PGI OpenACC compiler is an closed source software, we cannot get deep insights of its implementation. Instead, we used the AMD CodeXL profiling tool [AMD 2014] to analyze the generated OpenACC code on the AMD GPU. We discovered that the OpenACC version has significantly longer OpenCL kernel execution time compared to our approach. This may attribute to the lacking of array index transformations to reduce the non-coalesced memory accesses of OpenCL kernels. For some benchmarks, such as `lu` and `is`, the GPU gives no advantage and the OpenACC runtime does not dynamically choose computing devices and thus leads to poor performance. On average, our approach is 13x and 32x faster than the OpenACC implementation on the NVIDIA and AMD platforms respectively.

7.3. Comparison to Hand-coded Implementation

Figure 10 compares the generated OpenCL code to the hand-written SNU implementation [Seo et al. 2011]. This provides *independently* hand-written OpenCL implementations of the NAS parallel benchmarks. We selected the largest possible input size for each benchmark. To provide a fair comparison, the experiments were carried out on two platforms where the developers have tested the code: a NVIDIA GTX 480 GPU and an AMD Radeon HD6970 GPU.

The data show mixed results. For benchmarks `bt`, `sp` and `ft`, our automatically generated code outperforms the hand-written code. This is mainly due to the data restructuring performed by our compiler, including dynamic index re-ordering, which is especially important for benchmarks `bt` and `sp`. For `cg` and `ep` the speedups for both OpenCL code versions are similar but our predictive model outperforms SNU NPB on `cg` by selecting the right computing device (CPU) and using the OpenMP version.



(a) NVIDIA GTX480 (b) AMD HD6970

Fig. 10: Speedup of the SNU implementation, our generated OpenCL code and the predictive modeling based approach for the largest possible input size. This experiment was performed on two GPUs that the SNU developers have used for testing and evaluation.

On the remaining benchmarks, `is`, `lu` and `mg`, our generated code is not as good as the SNU implementation. The SNU version of `lu` uses a different algorithm than the original OpenMP code [Seo et al. 2011]. Their implementation uses a hyperplane algorithm which is much more suited for GPU execution. Changing the algorithm is out of the scope of our approach. For `is` the SNU implementation uses atomic operations to compute a histogram and a parallel prefix sum algorithm which is not exposed in the OpenMP code and is not supported by our current implementation. The code for `mg` works on highly irregular multi-dimensional data structures. In the generated code these data structures are flattened and indirection in each dimension is used to navigate through the data. The SNU implementation uses a different approach that requires a single level of indirection which leads to vastly improved performance. Nonetheless, our `ML` model is able to pick the right computing device for those benchmarks and the performance gap between the manual implementation and our predictive modeling based approach is not significant.

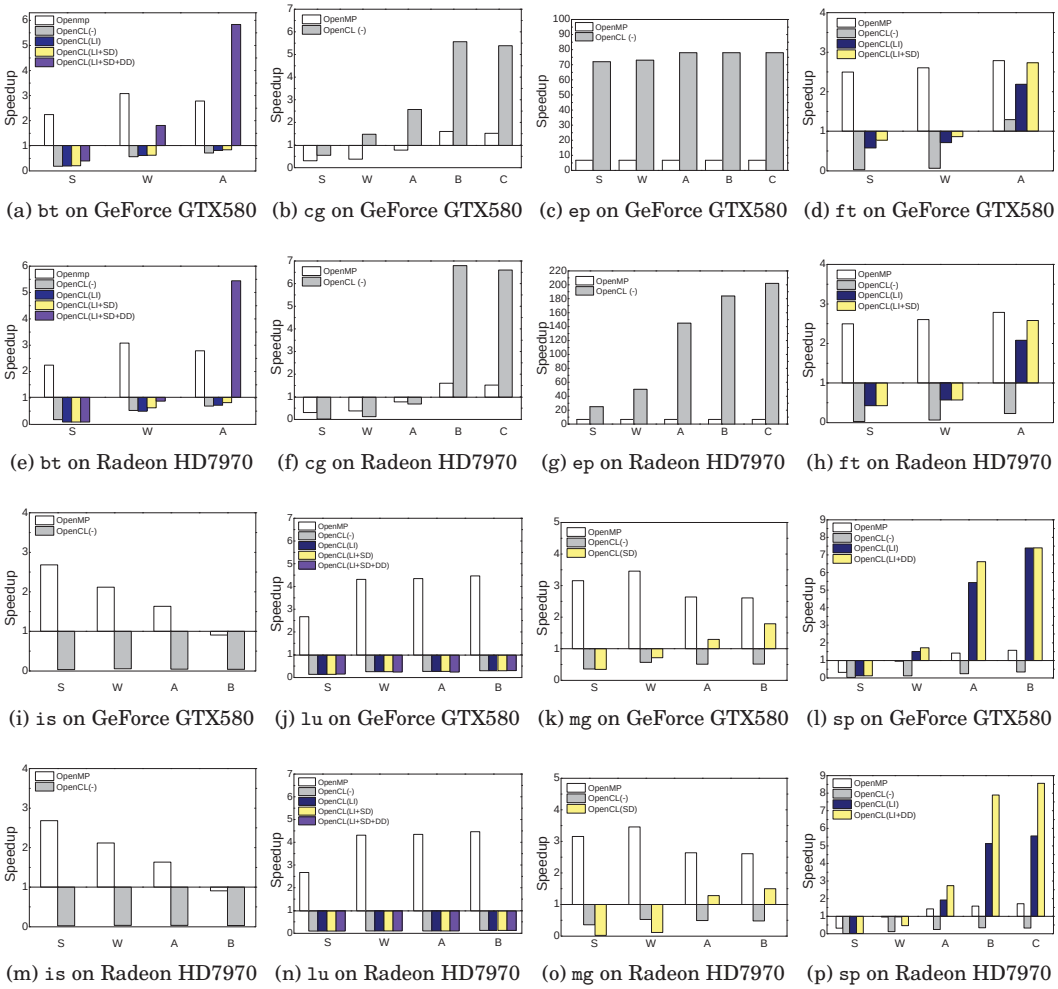
Overall our generated OpenCL code performs well. The hand-coded versions generally only perform better when algorithmic changes or code restructuring is performed which is difficult to be achieved by an automatic compiler without human involvement.

7.4. Performance Breakdown

Figure 11 shows the impact of the transformations described in Section 4 whenever they were applicable to a benchmark.

BT. As shown in Section 2 `bt` contains many multi-dimensional arrays and deeply nested loops. Without applying any of the transformations performance is thus poor on the GeForce system. But even when applying loop interchange and static data transformations the performance only improves marginally (up to 0.81x). This is because there are several, large sections of the program that require different data layouts to achieve memory coalescing and perform well on the GPU. Hence, when also applying dynamic array index re-ordering performance improves markedly. However, when compared to CPU performance we only see speedups for large input sizes W and A. On the Radeon system (Figure 11e) a similar behavior can be observed.

CG. This benchmark only contains one-dimensional arrays and is thus not amenable to loop interchange or data transformations. However, as the Figures 11b and 11f show, good performance on GPUs can already be achieved without any transformations. For



(a) bt on GeForce GTX580 (b) cg on GeForce GTX580 (c) ep on GeForce GTX580 (d) ft on GeForce GTX580

(e) bt on Radeon HD7970 (f) cg on Radeon HD7970 (g) ep on Radeon HD7970 (h) ft on Radeon HD7970

(i) is on GeForce GTX580 (j) lu on GeForce GTX580 (k) mg on GeForce GTX580 (l) sp on GeForce GTX580

(m) is on Radeon HD7970 (n) lu on Radeon HD7970 (o) mg on Radeon HD7970 (p) sp on Radeon HD7970

Fig. 11: Speedup over single-core execution on Corei7 (OpenMP) and both the NVIDIA GTX580 and AMD HD7970 systems (OpenCL). Where applicable we show the performance of the transformation steps: LI=loop interchange; SD=static data transformations; DD=dynamic data transformations.

small input sizes slowdowns can be observed due to insufficient amounts of computation. But for large input sizes the GPU outperforms the CPU with speedups of up to 5.56x and 6.79x on the two GPU architectures.

EP. Figures 11c and 11g show results for ep, an embarrassingly parallel benchmark that performs a significant amount of computation per work item. Its data structures are one-dimensional arrays and data transformations are thus not applicable. But even without any transformations speedups of up to 78x on the GeForce and 202x on the Radeon systems over single-core execution are observed. This compares to speedups of only up to 10x on the Corei7.

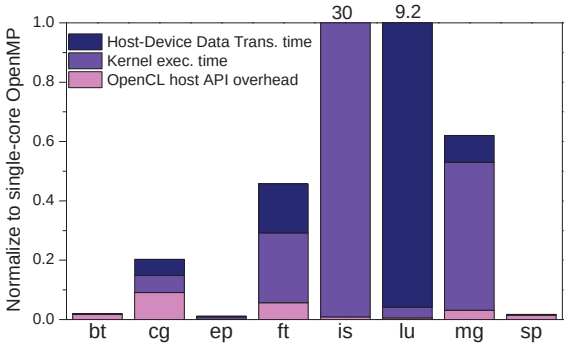


Fig. 12: OpenCL time breakdown on AMD HD7970. Each category is normalized to the sequential execution of the program (lower is better). This diagram includes the host-device data transfer time, OpenCL kernel time on the OpenCL host API overhead.

FT. For the `ft` benchmark we see speedups on the GPU of up to 2.6x for large input sizes (see figures 11d and 11h). With small inputs using the GPU leads to slowdowns, even after applying data transformations.

IS. On both architectures significant slowdowns are shown for GPU execution on all input sizes–from 0.03x to 0.78x on the GeForce GPU and from 0.01 to 0.04 on the Radeon GPU (Figures 11i and 11m). But even on the CPU only small speedups can be achieved. This is dominated by communication with the host memory. Furthermore, each OpenCL work-item needs to keep a private copy of a large array which means that the total number of work-items is limited by the GPU memory. This leads to underutilization of the GPU which further reduces performance. Data transformations are not applicable to this benchmark because it only works on one-dimensional arrays.

LU. Even though the OpenMP code shows good speedups on the CPUs, performance on GPUs is poor despite applying data transformations (Figures 11j and 11n). `lu` is a complex program and its OpenMP version exploits pipeline parallelism using a combination of asynchronous parallel loops and a bit-array to coordinate pipeline stages. The current SIMD-like execution models of GPUs are not designed to exploit this type of parallelism. As a result, large parts of `lu` can not utilize GPU's massive parallel processing units and have to be serialized which explains the slowdown on the GPUs (as can be seen from the expensive host-device communication shown in Figure 12).

MG. For this benchmark, we see that speedups are only achieved for large input sizes and after data transformations have been applied (Figures 11k and 11o). However, on none of the architectures does the GPU outperform the CPU.

SP. Similar to `bt` dynamic data realignment is required to unlock the best performance for benchmark `sp` (Figures 11l and 11p). Static data transformations cannot improve on the original data layout. Speedups of up to 8.6x on the GeForce GPU and up to 4.1x on the Radeon GPU are shown for large input sizes. This compares to speedups of up to 2.0x on the CPU. For small data sizes, however, the CPU outperforms the GPU.

7.5. Breakdown of OpenCL Runtime

Figure 12 shows the breakdown of OpenCL execution time for each benchmark (with the largest possible input size) on the AMD HD7970 system. This information was collected using the AMD CodeXL profiling tool [AMD 2014]. The time is normalized to single-core exeuction of the original OpenMP program (lower is better). Each stacked

	F1	F2	F3	F4
S	0.0020	0	0	0.81
W	0.0004	0	0	0.82
A	0.0003	0	0	0.82

(a) w/o data transformations

	F1	F2	F3	F4
S	0.0020	**0.731**	0	0.81
W	0.0004	**0.780**	0	0.82
A	0.0003	**0.999**	0	0.82

(b) w/ data transformations

Table IV: Features of `bt`

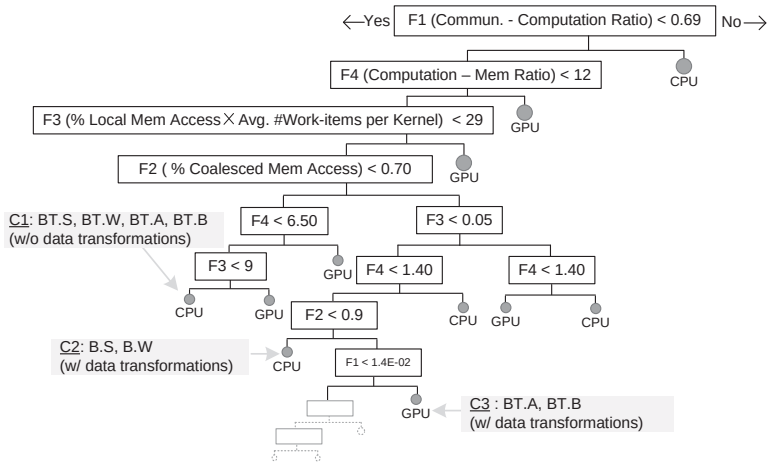


Fig. 13: The model used for `bt` on NVIDIA GeForce GTX 580. Predictions for `bt` with and without data transformations are marked as C1, C2 and C3.

bar consists of host-device data transfer time, cumulative OpenCL kernel execution time, and overhead for executing OpenCL host APIs the CPU.

Some benchmarks, e.g. `ep`, `bt`, `cg` and `sp`, can make effective use of the GPU with small kernel execution time and overall faster performance. Some benchmarks, by contrast, are not suitable for GPU execution. For example, for `is`, the small number of work-items leads to under-utilization of the GPU and because each GPU processing unit is simpler and weaker then a CPU core, this results in longer kernel execution time. For `lu`, the frequent CPU serial execution introduces intensive host-device communication and leads to slowdown performance. This figure shows that not all OpenCL programs can utilise the GPU and the available parallelism and the communication cost of task off-loading are important factors when determining which device to use to run the program.

7.6. Analysis of Predictive Models

Figures 13 and 14 show the decision trees constructed for the two systems by excluding `bt` from the training set. The learning algorithm automatically places the most relevant features at the root level and determines the architecture-dependent threshold for each node. All this is done automatically without the need of expert intervention.

As an example, the features for benchmark `bt` are shown in table IV.[4] We show the features both before and after applying data transformations according to the exam-

[4]The feature values for all benchmarks can be found at http://homepages.inf.ed.ac.uk/s0898672/cgo2013-features.tgz.

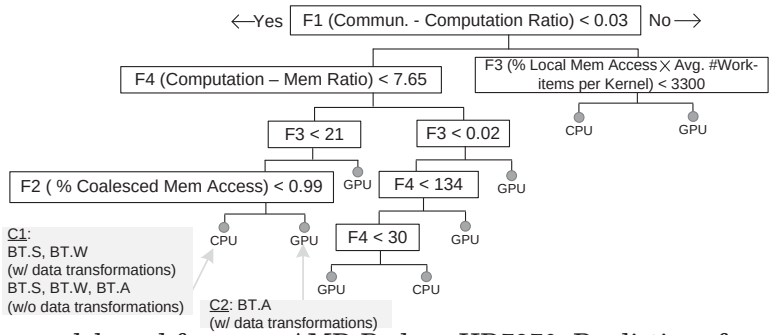


Fig. 14: The model used for bt on AMD Radeon HD7970. Predictions for bt with and without data transformations are marked as C1 and C2.

ple shown in section 2. This demonstrates the impact of the transformations on the mapping decision.

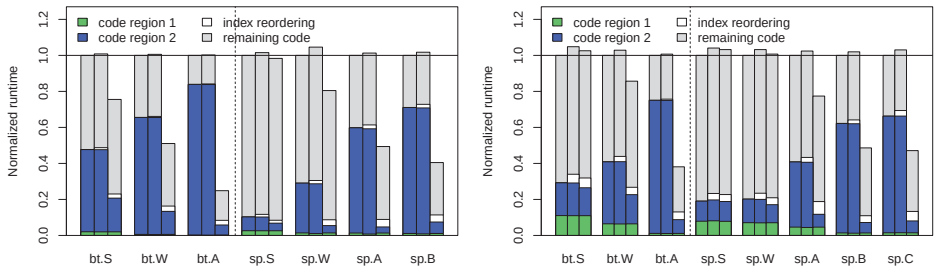
At the root of the tree in figure 13 we look at the value for the communication-computation ratio (F1). In all versions the value is far below the threshold. We thus proceed to the left subtree until reaching the fourth level of the tree. This node looks at the percentage of coalesced memory accesses (F2). Without data transformations none of the accesses are coalesced and the left branch is taken, eventually leading to CPU execution. With data transformations memory coalescing has been improved (see bold values in table IVb). For input sizes S and W the percentage of coalesced accesses is less than 80%. For A almost all accesses are coalesced due to dynamic index reordering (see section 4.1). All values are above the threshold so the right branch is taken. We follow the same branches until another node of F2 is reached. This time the threshold is higher, namely 0.9. For input sizes S and W the left branch is taken which leads to execution on the CPU. For the larger input size A we take the right branch and eventually reach a node predicting to run on the GPU. All programs get mapped to the right device.

Figure 14 shows the decision tree constructed for the AMD Radeon system. The first node also looks at the ratio between communication and computation. The threshold is lower (0.03), but for all versions of bt the ratio is still below the threshold. The same path is followed by all versions until the fourth level of the tree is reached. At this point we look at the percentage of coalesced accesses. The versions without data transformations are mapped to the CPU because none of the accesses are coalesced. Even when applying data transformations, for input sizes S and W the value is below the threshold and the code gets mapped to the CPU. Only input size A is mapped to the GPU. All programs are again mapped to the right device.

7.7. Dynamic Index Reordering

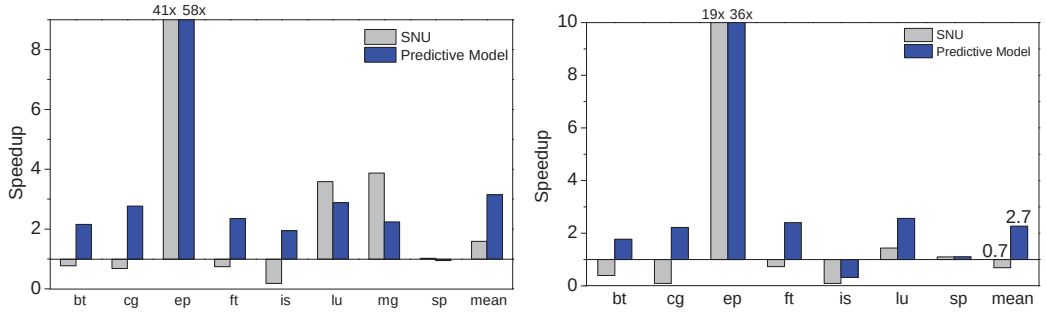
The benchmarks bt and sp contain candidate regions for dynamic index reordering. Figure 15 shows the performance of the benchmarks with different input sizes when applying dynamic index reordering to none, the first or the second of those regions. The performance is normalized to the runtime when the transformation is not applied. In each case the runtime is broken up into the runtimes for the two candidate code regions, the overhead of the transformation (if applicable) and the rest of the program.

The first candidate region makes up only a small fraction of the overall runtime of both benchmarks; 1-3% on the NVIDIA and 1-11% on the AMD system. When applying dynamic index reordering here the performance of this region barely improves, because there are not many memory accesses that benefit from the transformation. The cost of



(a) NVIDIA GeForce GTX580 (b) AMD Radeon HD7970

Fig. 15: Performance impact of dynamic index reordering when applying the transformation to none, the first or the second of the candidate regions. The runtime is broken up into the runtime for the two candidate code regions, the runtime of the transformation (if applicable) and the rest of the program.



(a) AMD Llano (b) Intel Ivy Bridge

Fig. 16: Speedup averaged across inputs of the manual SNU code and our model on systems with integrated GPUs.

reordering the data thus often outweighs the benefits which leads to minor slow-downs overall.

The second region, on the other hand, makes up a larger chunk of the overall runtime. Applying dynamic index reordering significantly reduces the runtime of this region. Since the overhead of data reordering is comparatively small big overall runtime reductions are achieved by applying the transformation to this region: up to 75% on the NVIDIA system and 62% on the AMD system.

Similar to predicting which device to run a program on, we also use decision trees to determine when dynamic index reordering is beneficial (see Section 5.4 for details). Applying this model for `bt` and `sp` achieves an accuracy of 79% on the NVIDIA and 94% on the AMD system.

7.8. Performance on Integrated Systems

GPU technology is constantly evolving. To check our approach also works on new devices we evaluated it on two systems with integrated GPUs, AMD Llano (A8-3850) and Intel Ivy Bridge (Core i5 3570K). Figure 16 shows a summary of the results using the SNU implementation and our predictive modeling approach. The manual SNU code only achieves speedups of 1.6x and 0.7x on average compared to 3.1x and 2.7x of our approach.

The integrated GPUs on these systems are less powerful than the discrete GPUs we evaluated before. This demonstrates even more the need for a model that only maps code to the GPU when it is beneficial. Integrated GPUs share the system memory with the CPU, making data movements between the devices cheaper or even unnecessary in the case of Intel IvyBridge. Because most benchmark in the NAS parallel benchmark suite are compute-intensive this advantage does not lead to improved performance overall. Nonetheless, our portable `ML` based approach is still able to achieve a speedup on average.

8. RELATED WORK

GPU Programming Languages. Programming support for GPUs has been a critical issue and CUDA has been a significant reason for the success of general-purpose computing on GPUs. Impressive performance has been achieved with orders of magnitude improvements found for certain applications [Ryoo et al. 2008]. Despite the popularity of CUDA, it mainly targets NVIDIA GPUs and is not directly portable to other GPUs or more general heterogeneous architectures. OpenCL has the promise of having more general applicability at the cost of a potentially more complex programming model. Due to the relative immaturity of language implementations recent work has focused on how to improve performance either using different code transformations [Lee and Eigenmann 2010] or partitioning across multiple GPUs [Kim et al. 2011]. The benchmarks considered are largely those well fitted to GPU architectures. In [Bordawekar et al. 2010; Grewe and O'Boyle 2011; Lee et al. 2010a] it was shown that while GPUs can often give significant performance for kernels, in some cases it is better not to use the GPU but use the multi-core instead.

High Level Programming Models. There have been several different approaches to generating GPU code from simpler higher level languages. In C to CUDA [Baskaran et al. 2010] nested loops are represented in a polyhedral framework which when mapped to the GPU give excellent performance. However, the set of programs that can be handled this way is extremely small and cannot be applied to more general parallel programs with arbitrary data dependence and control-flow structure. In Sponge [Hormati et al. 2011] again good performance is achieved. The benchmarks used were rewritten from the Parboil benchmarks, a CUDA benchmark suite, in StreamIt format so again the benchmarks are those that are well fitted to GPUs. Apart from Gregg et al. [Gregg et al. 2010], none of these approaches uses OpenCL and most of them rely on the user to provide separate kernel version for CPUs and GPUs. We circumvent this problem by automatically generating multi-versions of the input program and building a portable machine learning model to automatically select a code version at runtime.

Automatic Generation of GPU Programs. The OpenMPC compiler [Lee et al. 2009] translates OpenMP to CUDA programs. Unlike our approach OpenMPC does not perform dynamic data transformations nor use predictive modeling to select a code version across different GPU architectures. The OpenACC programming interface [OpenACC 2013] defines a set of compiler directives for expressing loop- and region-based parallelism. Using an OpenACC enabling compiler, the parallelism can be translated into OpenCL or CUDA implementations or be off-loaded onto an accelerator. In [Baskaran et al. 2010] CUDA programs are automatically generated from sequential, affine C programs using the polyhedral model. In all of the above approaches the code always gets executed on the GPU. Prior work on automatic generation of parallel GPU code from sequential programs also includes Par4ALL [Amini et al. 2012], PPCG [Verdoolaege et al. 2013] and [Wang et al. 2014a]. Unlike our approach, they do not consider the problem of selecting the most suitable device from the host CPU and the GPU to run the code.

Optimizing GPU Programs. CGCM [Jablin et al. 2011] is a CPU-GPU communication system to optimize CUDA applications between the host and the GPU. In the following work [Jablin et al. 2012], DyManD was proposed to overcome the limitation of CGCM by replacing static analysis with a dynamic runtime system. DyManD is able to optimize programs that can not automatically handle by CGCM. CUDA-lite [Ueng et al. 2008] relies on programmer annotations to translate exploit GPU performance by coalescing memory accessing. Sung et al. [Sung et al. 2010] propose a data layout transformation for structure grid programs (e.g. stencil code). The input to their tool are arrays that are in a restricted form. Dymaxion [Che et al. 2011] allows programmers to *manually* apply index reordering for CUDA programs. In contrast to Dymaxion which has a single data layout for the entire program, our compiler *automatically* applies dynamic index reordering to parts of the program when such a transformation is profitable. Furthermore, in Dymaxion index reordering can only be applied when transferring data from the host to the GPU, while our technique is applied when the data is already on the GPU. Recently, Kayiran et al. proposed a dynamic scheduling approach to determine the optimal number of GPU threads to reduce the resource contention on the GPU [Kayiran et al. 2013]. The StarPU runtime system provides a unified framework for scheduling numerical kernels on heterogeneous systems [Augonnet et al. 2011]. StarPU requires the developers to provide a cost model for each task and uses a heuristics to dynamically schedule parallel tasks. This fine-grained, runtime-based approach complements to our compiler-based approach.

Data Layout Transformation. Data layout crucial for application performance and there is an intensive body of work for data layout transformation on CPUs. A good review of existing CPU techniques could be found on Karlsson's report [Karlsson 2009]. DL is a runtime data layout transformation framework for GPU applications [Sung et al. 2012]. It offers a number of useful data layout transformations such as transforming array-of-structures to/from structure-of-arrays and in-place layout conversion. Prior work also includes Dymaxion[Che et al. 2011] and in-place matric transposition [Sung et al. 2014; Gustavson et al. 2012]. Impressive results have been achieved by those approaches. Unlike prior work that relies on the programmer to determine the cost and benefit of data transformations, our approach uses machine learning to automatically learn a cost model that automatically decide whether a transformation is beneficial for given program, input and hardware.

Mapping Parallel Programs. The majority of prior research on parallelism mapping has focused on building platform-specific heuristics [Ramanujam and Sadayappan 1989; Huang et al. 2009] for a certain class of platforms. Such an approach is tightly coupled to a specific architecture and as a result can not adapt to the fast evolving GPU architecture. Our predictive model, on the other hand, can adapt to the change of hardware and compilers by automatically learning from data. Some other approaches use iterative compilation and search to tune GPU programs [Datta et al. 2008]. These approaches, however, can lead to excessive profile runs for a single program. Our recent work [Grewe et al. 2013; Wen and O'Boyle 2014] takes the machine learning based approach further to automatically partitioning GPU kernels between the CPU and GPU in the presence of workload contention (i.e there are multiple programs compete for the shared computing resources). Our scheme achieves significant speedups over a GPU-only scheme, demonstrating the advantages of machine learning based parallelism mappings.

Predictive Modeling. In addition to optimizing sequential programs [Cooper et al. 1999], recent studies have shown that predictive modeling is effective in optimizing parallel programs [Wang and O'Boyle 2009, 2010; Collins et al. 2013; Wang and O'boyle

2013; Emani et al. 2013; Wang et al. 2014b] and scheduling parallel workload [Grewe et al. 2011]. The Qilin [Luk et al. 2009] compiler uses off-line profiling to create a regression model that is employed to predict a data parallel program's execution time. Unlike Qilin, our approach does not require any profiling runs during compilation. Recently, machine learning is used to predict the best mapping of a single OpenCL kernel [Grewe and O'Boyle 2011; Ogilvie et al. 2014]. In contrast to this work, our compiler automatically transforms large OpenMP programs into OpenCL-based programs and predicts whether the OpenMP or OpenCL code gives the best performance on the system.

9. CONCLUSION AND FUTURE WORK

This article has described a compilation approach that takes shared memory programs written in OpenMP and outputs OpenCL code targeted at GPU-based heterogeneous systems. The proposed approach uses loop and array transformations to improve the memory behavior of the generated code. OpenCL is a portable standard and we evaluate its performance on different platforms, NVIDIA GeForce and AMD Radeon discrete GPUs, and integrated GPUs. This approach was applied to the whole NAS parallel benchmark suite where we show that in certain cases the OpenCL code generated can produce significant speedups (up to 202x). However GPUs are not best suited for all programs and in some cases it is more profitable to use the host multi-core instead. We developed an approach based on machine learning that determines for each new program whether the multi-core CPU or the GPU is the best target. If the multi-core is selected we run the appropriate OpenMP code as it currently outperforms OpenCL on multi-cores. This model is learned on a per platform basis and we demonstrate that the model adapts to different platforms and achieves consistent prediction accuracy. We thus build on the portability of OpenCL as a language by developing a system that is performance portable as well.

Future work will examine a much greater range of program optimizations. In particular we wish to examine exploitation of the GPU memory hierarchy and apply auto-vectorization; both of these are likely to benefit the AMD Radeon and other GPUs.

References

AMD. 2013. AMD/ATI Stream SDK. http://www.amd.com/stream/. (2013).

AMD. 2014. CodeXL C Powerful Debugging, Profiling & Analysis. http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/codexl/. (2014).

Mehdi Amini, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François xavier Pasquier, Grégoire Péan, and Pierre Villalon. 2012. Par4All: From Convex Array Regions to Heterogeneous Computing. In *2nd International Workshop on Polyhedral Compilation Techniques (IMPACT 2012)*.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (2011), 187–198.

Muthu M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC '10*.

Rajesh Bordawekar, Uday Bondhugula, and Ravi Rao. 2010. Believe it or not!: multi-core CPUs can match GPU performance for a FLOP-intensive application!. In *PACT '10*.

Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. 2012. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science* 9, 0 (2012), 1910 – 1919.

Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *SC '11*.

Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. 2013. MaSiF: machine learning guided auto-tuning of parallel skeletons. In *HiPC*.

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *LCTES '99*.

Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU '10*.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08*.

Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *PLDI '04*.

M.K. Emani, Zheng Wang, and M.F.P. O'Boyle. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*.

Chris Gregg, Jeff Brantley, and Kim Hazelwood. 2010. *Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems*. Technical Report. Department of Computer Science, University of Virginia.

Dominik Grewe and Michael O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC'11*.

D. Grewe, Zheng Wang, and M.F.P. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO '13*.

Dominik Grewe, Zheng Wang, and Michael O'Boyle. 2013. OpenCL Task Partitioning in the Presence of GPU Contention. In *LCPC '13*.

Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2011. A Workload-aware Mapping Approach for Data-parallel Programs. In *HiPEAC '11*.

Fred Gustavson, Lars Karlsson, and Bo Kågström. 2012. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. *ACM Trans. Math. Softw.* (2012).

Amir Hormati, Mehrzad Samadi, Mark Woh, Trevor N. Mudge, and Scott A. Mahlke. 2011. Sponge: portable stream programming on graphics engines. In *ASPLOS XVI*.

Lei Huang, Deepak Eachempati, Marcus W. Hervey, and Barbara Chapman. 2009. Exploiting global optimizations for OpenMP programs in the OpenUH compiler. In *PPoPP '09*.

Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically managed data for CPU-GPU architectures. In *CGO '12*.

Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU communication management and optimization. In *PLDI '11*.

Lars Karlsson. 2009. *Blocked in-place transposition with application to storage format conversion*. Technical Report.

Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT '13*. 157–166.

Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. In *PPoPP '11*.

Jaekyu Lee, N.B. Lakshminarayana, Hyesoon Kim, and R. Vuduc. 2010b. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *MICRO 2010*.

Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC '10*.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09*.

Victor W. Lee and others. 2010a. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10*.

LLVM. 2013. The LLVM Compiler Infrastructure Project. http://llvm.org/. (2013).

John Lu and Keith D. Cooper. 1997. Register Promotion in C Programs. In *PLDI '97*.

Chi-keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO 42*.

Christos Margiolas and Michael F. P. O'Boyle. 2014. Portable and Transparent Host-Device Communication Optimization for GPGPU Environments. In *CGO '14*.

NVIDIA Corp. 2013. NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html. (2013).

William Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2014. Fast Automatic Heuristic Construction Using Active Learning. In *LCPC '14*.

OpenACC. 2013. The OpenACC Application Program Interface. http://www.openacc-standard.org/. (2013).
PathScale Inc. 2013. NPB2.3-OpenACC-C. https://github.com/pathscale/NPB2.3-OpenACC-C. (2013).
J. Ross Quinlan. 1993. *C4.5: programs for machine learning*.
J. Ramanujam and P. Sadayappan. 1989. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89*.
Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *PPoPP '08*.
Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IISWC '11*.
Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *PPoPP '12*.
Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *POPL '96*.
John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. 2010. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO '10*.
I-Jui Sung, Juan Gómez-Luna, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. 2014. In-place Transposition of Rectangular Matrices on Accelerators. In *PPoPP '14*.
I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. 2012. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*.
I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. 2010. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT '10*.
The Omini Compiler Project. 2009. NAS Paralel Benchmark v2.3 OpenMP C version. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html. (2009).
The Portland Group. 2010. PGI Fortran & C Accelator Programming Model. (2010). White Paper.
Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael O'Boyle. 2009. Towards a holistic approach to auto-parallelization. In *PLDI '09*.
Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. 2008. Languages and Compilers for Parallel Computing. Chapter CUDA-Lite: Reducing GPU Programming Complexity, 1–15.
University of Illinois at Urbana-Champaign UIUC. 2013. Parboil Benchmark Suite, http://impact.crhc.illinois.edu/parboil.php. (2013).
Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (2013).
Zheng Wang and Michael O'Boyle. 2009. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09*.
Zheng Wang and Michael O'Boyle. 2010. Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach. In *PACT '10*.
Zheng Wang and Michael F. P. O'boyle. 2013. Using Machine Learning to Partition Streaming Programs. *ACM Trans. Archit. Code Optim.* (2013).
Zheng Wang, Dainel Powel, Bjoern Franke, and Michael FP O'Boyle. 2014a. Exploitation of GPUs for the Parallelisation of Probably Parallel Legacy Code. In *CC '14*.
Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. 2014b. Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping. *ACM Trans. Archit. Code Optim.* (2014).
Yuan Wen, Zheng Wang, and Michael O'Boyle. 2014. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *HiPC '14*.
Michael Wolfe. 2010. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*.
Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *PLDI '10*.