Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems

Dominik Grewe Zheng Wang Michael F.P. O'Boyle
School of Informatics, University of Edinburgh
{dominik.grewe, zh.wang}@ed.ac.uk, mob@inf.ed.ac.uk

Abstract

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 paper presents a compiler based approach to automatically generate optimized OpenCL code from data-parallel OpenMP programs for GPUs. Such an approach brings together the benefits of a clear high level language (OpenMP) and an emerging standard (OpenCL) for heterogeneous multi-cores. A key feature of our scheme is that it leverages existing transformations, especially data transformations, to improve performance on GPU architectures and uses predictive modeling to automatically 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 two distinct GPU based systems: Core i7/NVIDIA GeForce GTX 580 and Core i7/AMD Radeon 7970. We achieved average (up to) speedups of 4.51x and 4.20x (143x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

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

General Terms Experimentation, Languages, Measurement, Performance

Keywords GPU, OpenCL, Machine-Learning Mapping

1. Introduction

Heterogeneous systems consisting of a host multi-core and 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 space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier for users.

High level shared memory programming languages such as OpenMP are more attractive. They give a simple upgrade path to parallelism for existing programs using pragmas. Although OpenMP is mainly used for programming shared memory multi-cores, it is a high-level language with little hardware specific information and can be targeted to other platforms. What we would like is the ease of programming of OpenMP with the GPU availability of OpenCL that is then optimized for a particular platform and gracefully adapts to GPU evolution. We deliver this by developing a compiler based approach that automatically generates optimized OpenCL from a subset of OpenMP. This allows the user to continue to use the same programming language, with no modifications, while benefiting automatically from heterogeneous performance.

The first effort in this direction is [17]. 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 paper data transformation are crucial to achieve good performance on GPUs. Secondly, the programs are always executed on GPUs. While GPUs

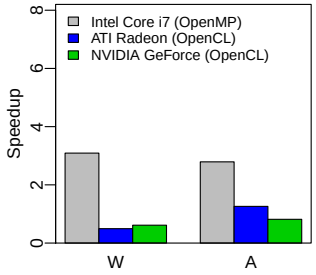
CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

```
#pragma omp parallel for
for (i=1;i<grid_points[0]-1;i++){
 for (j=1;j<grid_points[1]-1;j++){
  for (k=1;k<grid_points[2]-1;k++){
       ...
       lhs[i][j][k][0][0][0] = ...;
       lhs[i][j][k][0][0][1] = ...;
       ...
 } } }
```

(a) Original OpenMP

```
__kernel void lhsy_L1 (...) {
  int k = get_global_id (0) + 1;
  int j = get_global_id (1) + 1;
  int i = get_global_id (2) + 1;
  ...
  lhs[i][j][k][0][0][0] = ...;
  lhs[i][j][k][0][0][1] = ...;
  ...
  }
```

(b) Non-optimized OpenCL



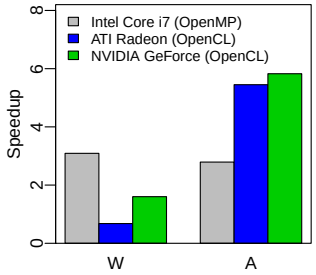
(c) Before optimizations

```
#pragma omp parallel for
for (i=1;i<grid_points[0]-1;i++){
 for (j=1;j<grid_points[1]-1;j++){
  for (k=1;k<grid_points[2]-1;k++){
       ...
       lhs[0][0][0][i][j][k] = ...;
       lhs[0][0][1][i][j][k] = ...;
       ...
 } } }
```

(d) Transformed OpenMP

```
__kernel void lhsy_L1 (...) {
  int k = get_global_id (0) + 1;
  int j = get_global_id (1) + 1;
  int i = get_global_id (2) + 1;
  ...
  lhs[0][0][0][i][j][k] = ...;
  lhs[0][0][1][i][j][k] = ...;
    ...
  }
```

(e) Optimized OpenCL



(f) After optimizations

Figure 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 multi-core CPU (f).

may deliver improved performance, they are not always superior to CPUs [4, 18]. A technique for determining when GPU execution is beneficial is needed. This paper addresses both of these issues and when evaluated on the full NAS parallel benchmarks our technique outperforms OpenMPC by a factor of 10.

A key feature of our scheme is that it uses predictive modeling 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 GPU evolution. 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.

This paper's technical contributions can be summarized as follows. It is the first to:

- automatically map all NAS parallel benchmarks, some of which are up to 3600 lines long with 66 kernels.
- use cost based dynamic data alignment for GPUs
- use predictive modeling to decide between different implementation languages on heterogeneous platforms
- outperform hand-coded OpenCL implementation

across different programs and architectures.

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 whether or not it is profitable to use a GPU 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 50 parallel loops potentially suitable for offloading to a GPU. Using our basic OpenMP to OpenCL translator yields the code shown in 1b. The parallel loop has been translated into a kernel where each of the loops is parallelized forming a 3D parallel work-item space 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 small W data size however gives disappointing performance when compared to executing the code on a multi-core as shown in figure 1c. If we execute the same code with a larger data 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 as shown in 1d, transforming the data layout of array `lhs`. This gives the new OpenCL program shown in 1e. Here the most rapidly varying indexes of the array correspond to the tile IDs giving coalesced memory accesses. 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.

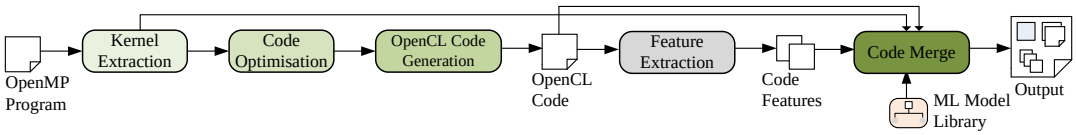


Figure 2: Compile time: Our compiler identifies kernels within the OpenMP program and performs several optimizations before generating the OpenCL code. This code is passed to our feature extraction tool to collect code features. In the final step the original OpenMP code, generated OpenCL code, code features and a machine learning (`ML`) model that is built off-line are merged into a single output program.

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 8, this decision varies from program to program and across different platforms and data sizes. 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. In the remainder of the paper we describe the translation and optimizations applied and develop a predictor that determines whether to exploit the GPU depending on circumstances.

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, the original OpenMP parallel loop and an optimized OpenCL kernel alternative. At runtime, a predictive model decides which version to use for execution. Our prototype compiler is implemented using Clang and LLVM.

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 of the generated code. These features are later used by the predictive model to select whether the OpenMP loop or OpenCL kernel version is best (see section 6). The features, together with the generated OpenCL code, the original OpenMP code and a machine learning (`ML`) predictor built off-line (see section 5) are merged into a single output program source.

Run-Time At execution, the generated program first updates the parameterized features based on the runtime values of parameters and passes the updated features to the `ML` model. The built-in model then predicts where to run the program and to pick either the OpenMP code for the multi-core CPU or the OpenCL code for the GPU.

Evaluating the model at runtime involves on the order of tens of operations and is thus negligible.

4. Code Generation and Optimization

Our framework currently converts OpenMP parallel loops, i.e. loops that are annotated with `omp for` or `omp for reduction`, into OpenCL kernels. A standard two-stage algorithm [2] is used to translate a parallel reduction loop. Other parallel OpenMP directives associated with task parallelism are not currently supported. Each parallel `omp` loop is translated to a separate kernel using the OpenCL APIs where each iterator is replaced by a global work-item ID.

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 the OpenMP version of the loop. Each generated program has a prediction function that decides where the code is to run and set the function pointers to the corresponding version. This is described in section 5.

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.

4.1 OpenCL Code Optimization

Our compiler performs a number of optimizations to improve the performance of the OpenCL code on the GPU. They 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 a classical loop dependence algorithm [14] 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. An example of this transformation was shown in figure 1: $[i, j, k, 0, 0, 0] \mapsto [0, 0, 0, i, j, k]$.

Memory Load Reordering In the original OpenMP programs, some accesses of read-only buffers can be reordered to form a sequence of consecutive load operations which can be vectorized. Our compiler automatically detects those candidates and replaces scalar load operations with an OpenCL

vector load. This can improve the memory performance of the generated OpenCL code.

Register Promotion On many occasions, a global memory scalar variable (or array) is accessed multiple times by a single OpenCL kernel. To reduce global memory latencies, our tool automatically creates a private register object for such variables. It generates code to load the data from the global memory to the private register copy (or write back to the global memory from the register for the last store operation). This allows the generated OpenCL kernel to reuse the object in the private register multiple times and the global memory operation only needs to be performed once.

Prefetching and Local Memory Optimization Our compiler automatically identifies read-only buffers that are used by multiple GPU threads and generates code to prefetch those buffers to local memory. Exploiting local memory reduces the memory latencies for GPU kernels.

4.1.1 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 [5].

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. Within the code region 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 6 describes a data-driven approach that solves this problem.

5. Predicting the Mapping

A crucial part of our approach is to automatically determine the best location 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 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 architecture 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.

Our model is a decision tree classifier where at every node of the tree a decision is made whether to follow the left or the right child. We used the C4.5 algorithm [22] to automatically construct the decision tree from training data by correlating the features to the best device. Section 8.4 shows an example of such a decision tree and its evaluation.

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)` × `avgws`	ratio local to global mem accesses × avg. # work-items per kernel
F4: `comp/mem`	computation-mem ratio

(b) Combinations of raw features

Table 1: List of code features.

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 program structure and runtime parameters. The static analysis characterizes a kernel as a fixed vector of real values, or *features* (see below).

5.1 Training the Predictor

The training process involves the collection of training data which is used to fit the model to the problem at hand. In our case 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 1a). 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. Double precision floating point operations are given a higher weight (4x) than single precision operations. 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.

Instead of using raw features, we group several features to form combined normalized features that carry more information than their parts (as shown in table 1b).

5.2.1 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 [7], Parboil [21], NVIDIA CUDA SDK [20] and AMD Accelerated Parallel Processing SDK [2]. 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 7.2).

5.3 Runtime Deployment

Once we have built the `ML` model as described above, we can insert the model together with the 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 Features As some loop bounds are dependent on the tasks' input data, the compiler may not be able to determine the value of certain features. In this case our compiler represents these features with static symbolic precomputation of loop bound variables. At execution time, these features are updated using runtime values.

Version Selection The first time a kernel is called the built-in predictive model selects a code version for execution. It uses updated features to predict the best device to run the program and sets the function pointer of each parallel loop 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.

6. A Model for Dynamic Index Reordering

In section 4.1.1 we described the dynamic index reordering transformation. This transformation can greatly improve performance on the GPU but it can also lead to slow-downs 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 use micro benchmarks to obtain the training data for this problem.[1] Given a simple kernel accessing an array in a non-coalesced way we vary the array size and the number of times the kernel is called, thereby changing the number of accesses to the array. 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.

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

	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 2: Hardware platform

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 reordered. At runtime one of them gets picked by the model.

7. Experimental Methodology

7.1 Experimental Setup

Platforms We evaluate our approach on two CPU-GPU systems: both use an Intel Core i7 6-core CPU. One system contains an NVIDIA GeForce GTX 580 GPU, the second an AMD Radeon 7970. Both run with the Ubuntu 10.10 64-bit OS. Table 2 gives detailed information on our platforms.

Benchmarks All eight of the NAS parallel benchmarks (v2.3) [1] were used for evaluation. Unlike many GPU benchmarks that are *single* precision, all the benchmarks except `is` are *double* precision programs.

7.2 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.

We use *leave-one-out cross-validation* to train and evaluate our machine-learning model for predicting the mapping. 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 machine-learning 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.

8. Experimental Results

In this section we evaluate our approach on two separate 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 a manual OpenCL implementation of the NAS benchmark suite [23] and OpenMPC [17], the closest related prior work, showing performance improvements of 1.4x and 10x respectively. We then take a closer look at the predictive model

and the dynamic index reordering transformation. Finally we present a brief evaluation of our approach on two heterogeneous systems with integrated GPUs.

8.1 Performance Evaluation

Figures 3 and 4 show speedups for the NAS benchmarks on the two heterogeneous systems described in section 7. 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 multi-core CPU[2] and 1.19x and 0.71x on the GPU.

The results show that speedups vary dramatically between 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 [24].

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".

[2] 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.

8.2 Comparison to State-of-the-Art

We compared our approach to two others: OpenMPC [17], a compiler translating OpenMP to CUDA, and the SNU NPB suite [23] which provides *independently* hand-written OpenCL implementations of the NAS parallel benchmarks. The results are shown in figure 5. Because OpenMPC generates CUDA code, we only evaluated it on the NVIDIA platform. We were unable to generate code for benchmarks `is`, `lu` and `mg` using OpenMPC.

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

The SNU implementation shows significantly better performance than OpenMPC but it is only able to outperform our approach on `ft` and `mg` on the NVIDIA platform and only on `ft` on the AMD platform. On average it achieves a speedup of 3.01x on NVIDIA and 2.02x on AMD compared to 4.18x and 4.22x of our predictive modeling approach. There are two reasons why we are able to outperform the hand-written versions of the benchmarks. Firstly, we perform aggressive data restructuring, including dynamic index reordering, which is especially important for benchmarks `bt` and `sp`. Secondly, we are able to detect when using the CPU over the GPU is the better option. So while the SNU code may be faster than the OpenCL code generated by our approach for some benchmarks (as we will show in the next section), we are often able to provide better performance by choosing *not* to use the GPU but the CPU.

We also tried to evaluate the benchmarks using the OpenACC standard. However, some OpenMP features, e.g. thread-private variables, are not supported by OpenACC and the compiler infrastructure is not very mature. Substantial effort in rewriting the code as well as more mature compilers are required to perform a fair comparison.

8.3 OpenCL Code Performance Comparison

Figure 6 compares the generated OpenCL code to the hand-written SNU implementation. We factor out the predictive model and focus solely on the quality of the generated OpenCL code. We selected the largest possible input size for each benchmark.

The data show mixed results. For benchmarks `bt` and `sp` our code outperforms the hand-written code. This is mainly due to the data restructuring by our compiler. For `cg`, `ep` and

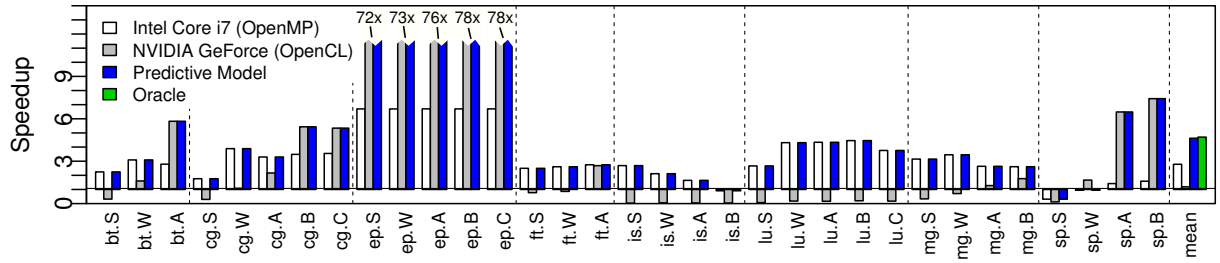


Figure 3: Performance of OpenMP on Intel CPU, OpenCL on NVIDIA GeForce GPU and the version selected by our predictive model. The predictive model outperforms the CPU-only approach by 1.69x and the GPU-only approach by 3.9x.

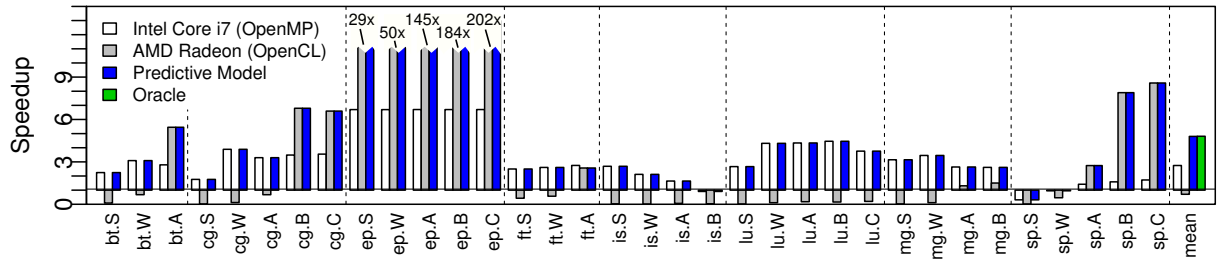
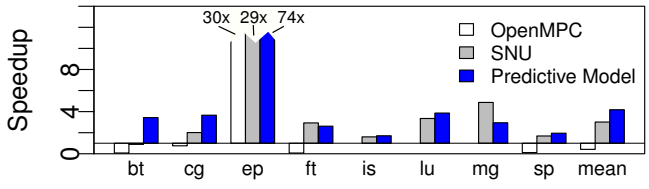
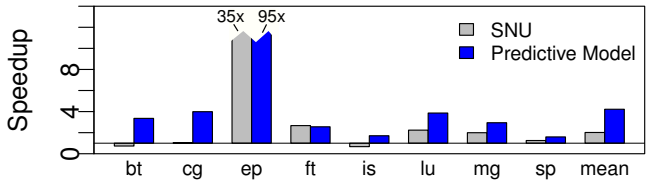


Figure 4: Performance of OpenMP on Intel CPU, OpenCL on AMD Radeon GPU and the version selected by our predictive model. The predictive model outperforms the CPU-only approach by 1.76x and the GPU-only approach by 6.8x.



(a) NVIDIA GeForce



(b) AMD Radeon

Figure 5: Speedup averaged across inputs of the OpenMPC compiler (NVIDIA only), the manual SNU implementation of the NAS parallel benchmarks and our predictive model.

`ft` the speedups are similar. 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 [23, Section 3.1]. Their implementation uses a hyperplane algorithm which is much more suited for GPU execution. Changing the algorithm is obviously 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. Theses types of optimizations are beyond the scope of a compiler. 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. For `ft`, the SNU code removes the array boundary checks in the primary reduction loop because the programmer knows it is safe to do so for certain inputs. This results in better GPU performance as branch overhead is eliminated.

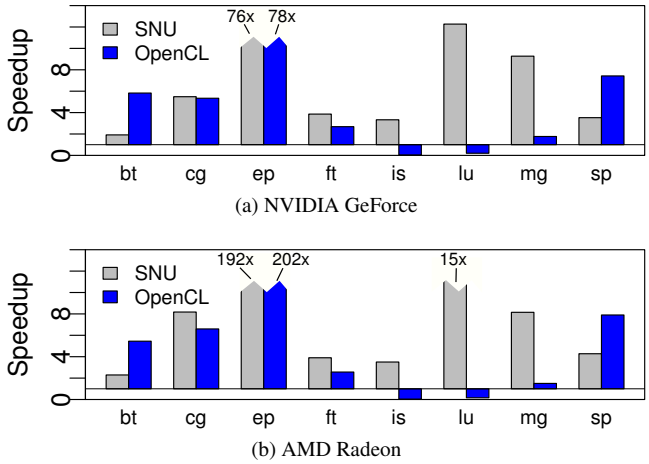
Overall our generated OpenCL code performs well. The hand-coded versions generally only perform better when algorithmic changes or code restructuring is performed which is beyond the scope of a compiler.

8.4 Analysis of Predictive Model

Figures 7 and 8 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 3.[3] We show the features both before and after applying data transformations according to the example shown in section 2. This demonstrates the impact of the transformations on the mapping decision.

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



(a) NVIDIA GeForce

(b) AMD Radeon

Figure 6: Speedup of the SNU implementation and our generated OpenCL code for the largest possible input size.

At the root of the tree in figure 7 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 3b). 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 4.1.1). 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 8 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.

8.5 Dynamic Index Reordering

The benchmarks `bt` and `sp` contain candidate regions for dynamic index reordering. Figure 9 shows the performance of the benchmarks with different input sizes when applying

	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 3: Features of `bt`

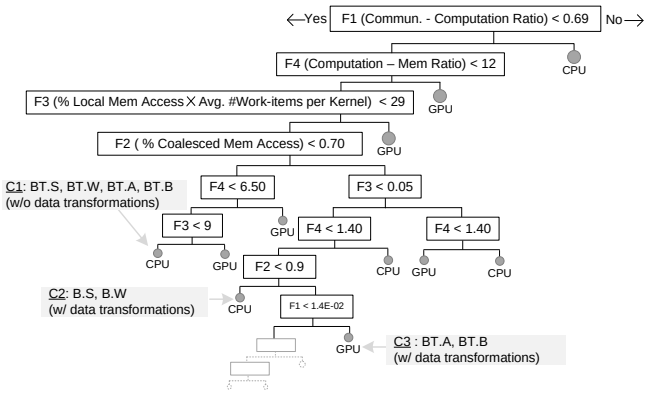


Figure 7: The model used for `bt` on NVIDIA GeForce. Predictions for `bt` with and without data transformations are marked as C1, C2 and C3.

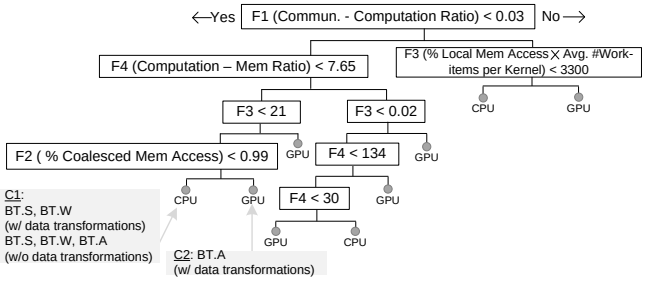
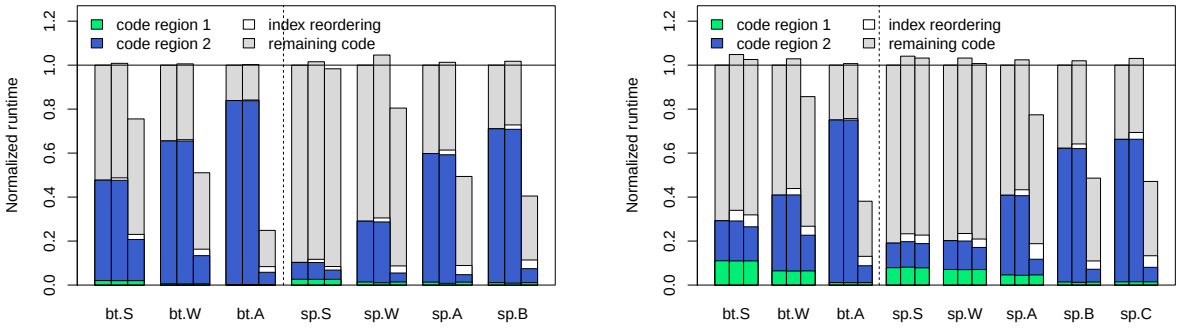


Figure 8: The model used for `bt` on AMD Radeon. Predictions for `bt` with and without data transformations are marked as C1 and C2.

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

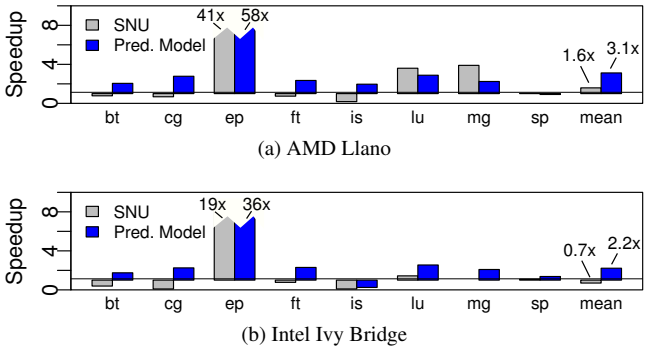
The second region, on the other hand, makes up a larger chunk of the overall runtime. Applying dynamic index reordering significantly reduces the runtime of this region. Since the overhead of data reordering is comparatively small big overall runtime reductions are achieved by applying the



(a) NVIDIA GeForce

(b) AMD Radeon

Figure 9: 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 runtimes 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

Figure 10: Speedup averaged across inputs of the manual SNU code and our model on systems with integrated GPUs.

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 6 for details). Due to space constraints we cannot show the decision trees but applying this model for `bt` and `sp` achieves an accuracy of 79% on the NVIDIA and 94% on the AMD system.

8.6 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 10 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 2.2x 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.

9. Related Work

Automatic Generation of GPU Programs No work has targeted automatic generation of OpenCL code from OpenMP programs. The OpenMPC compiler [17] is the nearest work, which 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.In [3] 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. Dubach et al. present a Java-compatible language called Lime for heterogeneous systems [8]. The GPU back end of the Lime compiler generates OpenCL code from the high level Lime languages. 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.

Programming Frameworks for GPUs Several programming models [10, 11, 15] have been proposed for GPU programming. These approaches provide APIs to develop GPU applications. All these approaches implicitly assume the GPU execution gives the best performance.

Optimizing GPU Programs Most of the prior research targets on optimizing CUDA programs. CGCM [13] is a CPU-GPU communication system to optimize CUDA applications between the host and the GPU. In the following work [12], 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. Dymaxion [5] 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. Further-

more, 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.

Predictive Modeling In addition to optimizing sequential programs [6], recent studies have shown that predictive modeling is effective in optimizing parallel programs [25, 26]. The Qilin [19] 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 [9]. 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.

10. Conclusion

This paper 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 two different platforms, NVIDIA GeForce and AMD Radeon. 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. Using our approach we achieve 1.39x and 2.09x improvement over a hand-coded, GPU-specific implementation. We thus build on the portability of OpenCL as a language by developing a system that is performance portable as well.

References

[1] NAS parallel benchmarks 2.3, OpenMP C version. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

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

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *CC '10*.

[4] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or not!: multi-core CPUs can match GPU performance for a FLOP-intensive application! In *PACT '10*.

[5] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *SC '11*.

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99*.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU '10*.

[8] C. Dubach, P. Cheng, R. Rabbah, D. Bacon, and S. Fink. Compiling a high-level language for GPUs (via language support for architectures and compilers). In *PLDI '12*.

[9] D. Grewe and M. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *CC '11*.

[10] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU '09*.

[11] A. Hormati, M. Samadi, M. Woh, T. N. Mudge, and S. A. Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS '11*.

[12] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *CGO '12*.

[13] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *PLDI '11*.

[14] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002.

[15] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple GPUs. In *PPoPP '11*.

[16] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Takizawa. Evaluating performance and portability of OpenCL programs. In *Workshop on Automatic Performance Tuning 2010*.

[17] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09*.

[18] 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. In *ISCA '10*.

[19] C.-k. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO '09*.

[20] NVIDIA Corp. NVIDIA CUDA. `http://developer.nvidia.com/object/cuda.html`.

[21] U. of Illinois at Urbana-Champaign. Parboil benchmark suite, `http://impact.crhc.illinois.edu/parboil.php`.

[22] J. R. Quinlan. *C4.5: programs for machine learning*. 1993.

[23] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *IISWC '11*.

[24] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization. In *PLDI '09*.

[25] Z. Wang and M. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09*.

[26] Z. Wang and M. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*.