Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping

Chi-Keung Luk
Software Pathfinding and Innovations
Software and Services Group
Intel Corporation
Hudson, MA 01749
chi-keung.luk@intel.com

Sunpyo Hong
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
shong9@gatech.edu

Hyesoon Kim
College of Computing
School of Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
hyesoon@cc.gatech.edu

ABSTRACT

Heterogeneous multiprocessors are increasingly important in the multi-core era due to their potential for high performance and energy efficiency. In order for software to fully realize this potential, the step that maps computations to processing elements must be as automated as possible. However, the state-of-the-art approach is to rely on the programmer to specify this mapping manually and statically. This approach is not only labor intensive but also not adaptable to changes in runtime environments like problem sizes and hardware/software configurations. In this study, we propose *adaptive mapping*, a fully automatic technique to map computations to processing elements on a CPU+GPU machine. We have implemented it in our experimental heterogeneous programming system called *Qilin*. Our results show that, by judiciously distributing works over the CPU and GPU, automatic adaptive mapping achieves a 25% reduction in execution time and a 20% reduction in energy consumption than static mappings on average for a set of important computation benchmarks. We also demonstrate that our technique is able to adapt to changes in the input problem size and system configuration.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Parallel Processors*; D.1.3 [**Software**]: Programming Techniques—*Parallel Programming*

General Terms

Performance

Keywords

Multicore, heterogeneous, GPU, adaptive, mapping, dynamic compilation.

1. INTRODUCTION

Multiprocessors have emerged as mainstream computing platforms nowadays. Among them, an increasingly popular class are those with *heterogeneous architectures*. By providing processing elements (PEs)[1] of different performance/energy characteristics on the same machine, these architectures could deliver high performance and energy efficiency [14]. The most well-known heterogeneous architecture today is probably the IBM/Sony Cell architecture, which consists of a Power processor and eight synergistic processors [26]. In the personal computer (PC) world, a desktop now has a multicore CPU and a GPU, exposing multiple levels of hardware parallelism to software, as illustrated in Figure 1.

In order for mainstream programmers to fully tap into the potential of heterogeneous multiprocessors, the step that maps computations to processing elements must be as *automated* as possible. Unfortunately, the state-of-the-art approach [16, 22, 36] is to rely on the programmer to perform this mapping *manually*: for the Cell, O'Brien *et al.* extend the IBM XL compiler to support OpenMP on this architecture [22]; for commodity PCs, Linderman *et al.* propose the Merge framework [16], a library-based system to program CPU and GPU together using the map-reduce paradigm. In both cases, the computation-to-processor mappings are *statically* determined by the programmer. This manual approach not only puts burdens on the programmer but also is *not adaptable*, since the optimal mapping is likely to change with different applications, different input problem sizes, and different hardware/software configurations.

In this paper, we address this problem by introducing a *fully automatic* approach that decides the mapping from computations to processing elements using *run-time adaptation*. We have implemented our approach into *Qilin*[2], our experimental system for programming heterogeneous multiprocessors. Experimental results demonstrate that our automated adaptive mapping performs nearly as well as manual mapping in terms of both execution time and energy consumption, while at the same time can tolerate changes in input problem sizes and hardware/software configurations.

The rest of this paper is organized as follows. In Section 2, we use a case study to further motivate the need of adaptive mapping. Section 3 then describes the Qilin system in details. We will focus on our runtime adaptation techniques in Section 4. Experimental evidences will be given in Section 5. Finally, we relate our works to others in Section 6 and conclude in Section 7.

[1]The term *processing element* (PE) is a generic term for a hardware element that executes a stream of instructions.
[2]"Qilin" is a mythical chimerical creature in the Chinese culture; we picked it as the project name for the heterogeneity of this creature.

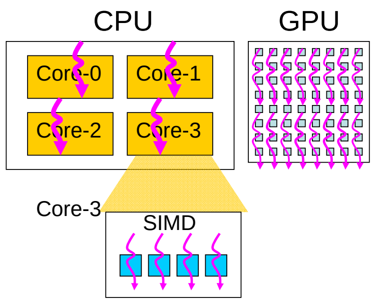


Figure 1: Multiple levels of hardware parallelism exposed to software on current CPU+GPU systems (The GPU has tens to hundreds of special-purpose cores, while the CPU has a few general-purpose cores. Within each CPU core, there is short-vector parallelism provided by the SIMD extension of the ISA.)

2. A CASE STUDY: WHY DO WE NEED ADAPTIVE MAPPING?

We now motivate the need of adaptive mapping with a case study on matrix multiplication, a very commonly used computation kernel in scientific computing. We measured the parallelization speedups on matrix multiplication with a heterogeneous machine consisting of an Intel multicore CPU and a Nvidia multicore GPU (details of the machine configuration are given in Section 5.1).

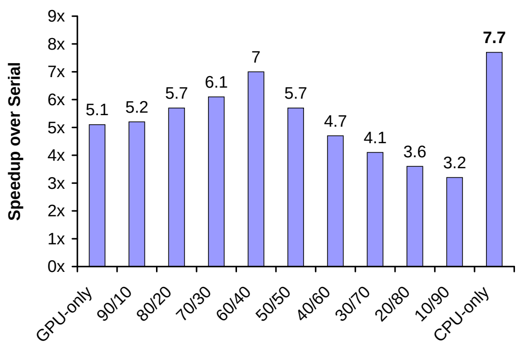
We did three experiments with different input matrix sizes and the number of CPU cores used. In each experiment, we varied the distribution of work between the CPU and GPU. For matrix multiplication $C = A * B$, we first divide A into two smaller matrices A_1 and A_2 by rows. We then compute $C_1 = A_1 * B$ on the CPU and $C_2 = A_2 * B$ on the GPU in *parallel*. Finally, we obtain C by combining C_1 and C_2. We use the best matrix multiplication libraries available: for the CPU, we use the Intel Math Kernel Library (MKL) [11]; for the GPU, we use the CUDA CUBLAS library [20].

Figure 2 shows the results of these three experiments. All input matrices are $N * N$ square matrices. The y-axis is the speedup over the serial case. The x-axis is the distribution of work across the CPU and GPU, where the notation "X/Y" means X% of work mapped to the GPU and Y% of work mapped to the CPU. At the two extremes are the cases where we schedule all the work on either the GPU or CPU.

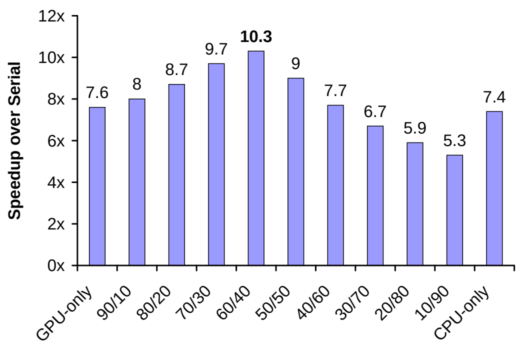
In Experiment 1, we use a relatively small problem size ($N = 1000$) with eight CPU cores. The low computation-to-communication ratio with this problem size renders the GPU less effective. As a result, the optimal mapping is to schedule all work on the CPU. In Experiment 2, we increase the problem size to $N = 6000$ and keep the same number of CPU cores. Now, with a higher computation-to-communication ratio, the GPU becomes more effective—both the GPU-alone and the CPU-alone speedups are over 7x. And the optimal mapping is to schedule 60% of work on the GPU and 40% on the CPU, resulting into a 10.3x speedup. In Experiment 3, we keep the problem size to $N = 6000$ but reduce the number of CPU cores to only two. With much less CPU horsepower, the CPU-only speedup is limited to 2x and the optimal mapping now is to schedule 80% of work on the GPU and 20% on the CPU.

It is clear from these experiments that even for a single application, the optimal mapping from computations to processing elements highly depends on the input problem size and the hardware capability. Needless to say, different applications would have

(a) **Experiment 1 ($N = 1000$, #CPU Cores = 8)**



(b) **Experiment 2 ($N = 6000$, #CPU Cores = 8)**



(c) **Experiment 3 ($N = 6000$, #CPU Cores = 2)**

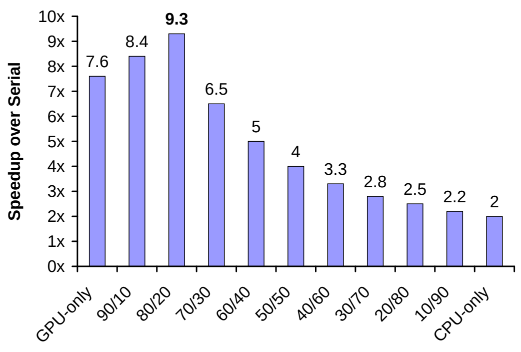


Figure 2: Matrix multiplication experiments with different input matrix sizes and number of CPU cores used. The input matrices are N by N. The notation "X/Y" on the x-axis means X% of work mapped to the GPU and Y% of work mapped to the CPU.

different optimal mappings. Therefore, we believe that any *static* mapping techniques would not be satisfactory. What we want is a *dynamic* mapping technique that can automatically adapt to the runtime environment, as we are going to propose next.

3. THE QILIN PROGRAMMING SYSTEM

Qilin is a programming system we have recently developed for heterogeneous multiprocessors. Figure 3 shows the software architecture of Qilin. At the application level, Qilin provides an API to programmers for writing *parallelizable* operations. By explicitly expressing these computations through the API, the compiler is alleviated from the difficult job of extracting parallelism from serial code, and instead can focus on performance tuning. Simi-

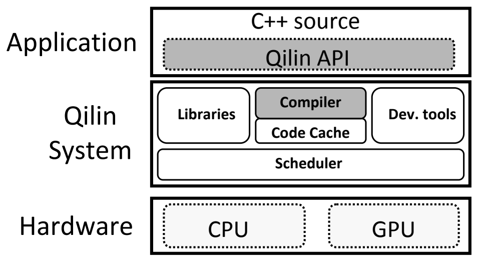


Figure 3: Qilin software architecture

Category	Operations
Construct /Convert	Create1D(), Create2D(), ToNormalArray()
Elementwise	+, -, *, /, Max, Min, And, Or, <, <=, ==, >, >=, Sin, Cos, Sqrt, CondSelect
Reduction	+, *, Max, Min
Transformation	Shift, Rotate, Transpose, Replicate
Linear Algebra	BLAS 1-3
Utility	Random number generator, Cumulative normal distribution

Figure 4: Qilin Stream-API operations.

```
float qilin_pi()  {
    const float N = 100000; // total number of random points
    float X[N], Y[N];

    // Generate N random points within the unit square
    GenerateRandomFloats(X, N, 0.0, 1.0);
    GenerateRandomFloats(Y, N, 0.0, 1.0);

    // Create Qilin-arrays Qx and Qy from C-arrays X and Y
    QArrayF32 Qx = QArrayF32::Create1D(N, X);
    QArrayF32 Qy = QArrayF32::Create1D(N, Y);

    // For each random point, determine if it is within
    // the quadrant
    QArrayF32 DistFromZero = Sqrt(Qx*Qx + Qy*Qy);
    QArrayF32 WithinQuadrant = (DistFromZero <= 1.0f);

    // Sum up the total number of random points within
    // the quadrant
    QArrayF32 S = Sum(WithinQuadrant);

    // Convert the result stored in Qilin-array S back
    // to C-array M
    float M[1];
    S.ToNormalArray(M, sizeof(float));

    // Compute Pi
    const float pi = 4.0 * (M[0] / N);
    return pi;
}
```

Figure 5: Computing the value of π with Qilin Stream API.

lar to OpenMP, the Qilin API is built on top of C/C++ so that it can be easily adopted. But unlike standard OpenMP, where parallelization only happens on the CPU, Qilin can exploit the hardware parallelism available on both the CPU and GPU.

Beneath the API layer is the Qilin system layer, which consists of a dynamic compiler and its code cache, a number of libraries,

```
void CpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
     Pixel* src_cpu = qSrc.NormalArray();
     Pixel* dst_cpu = qDst.NormalArray();
     int height_cpu = qSrc.DimSize(0);
     int width_cpu = qSrc.DimSize(1);

     // … Filter implementation in TBB …
}
void GpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
     Pixel* src_gpu = qSrc.NormalArray();
     Pixel* dst_gpu = qDst.NormalArray();
     int height_gpu = qSrc.DimSize(0);
     int width_gpu = qSrc.DimSize(1);

     // … Filter implementation in CUDA …
}
void MyFilter(Pixel* src, Pixel* dst, int height, int width) {
     // Create Qilin arrays from normal arrays
     QArray<Pixel> qSrc = QArray<Pixel>::Create2D(height,
                                                  width, src);
    QArray<Pixel> qDst = QArray<Pixel>::Create2D(height,
                                                 width, dst);

     // Define myFilter as an operation that glues
     // CpuFilter() and GpuFilter()
     QArrayOp myFilter = MakeQArrayOp("myFilter",
                                      CpuFilter, GpuFilter);

     // Build the argument list for myFilter.
     // QILIN_PARTITIONABLE means the associated computation
     // can be partitioned to run on both CPU and GPU.
     QArrayOpArgsList argList;
     argList.Insert(qSrc, QILIN_PARTITIONABLE);
     argList.Insert(qDst, QILIN_PARTITIONABLE);

     // Apply myFilter with argList using the default mapping
     QArray<BOOL> qSuccess = ApplyQArrayOp(myFilter, argList,
                                           PE_SELECTOR_DEFAULT);

     // Convert from qSuccess[] to success
     BOOL success;
     qSuccess.ToNormalArray(&success, sizeof(BOOL));
}
```

Figure 6: Image filter written with the Threading-API approach.

a set of development tools, and a scheduler. The compiler dynamically translates the API calls into native machine codes. It also decides the near-optimal mapping from computations to processing elements using an adaptive algorithm. To reduce compilation overhead, translated codes are stored in the code cache so that they can be reused without recompilation whenever possible. Once native machine codes are available, they are scheduled to run on the CPU and/or GPU by the scheduler. Libraries include commonly used functions such as BLAS and FFT. Finally, debugging, visualization, and profiling tools can be provided to facilitate the development of Qilin programs.

Current Implementation.

Qilin is currently built on top of two popular C/C++ based threading APIs: The Intel Threading Building Blocks (TBB) [30] for the CPU and the Nvidia CUDA [21] for the GPU. Instead of directly generating CPU and GPU native machine codes, the Qilin compiler generates TBB and CUDA source codes from Qilin programs on the fly and then uses the system compilers to generate the final machine codes. We use CUDA drivers to communicate data between the CPU and GPU. The "Libraries" component in Figure 3 is implemented as wrapper calls to the libraries provided by the vendors: Intel MKL [11] for the CPU and Nvidia CUBLAS [20] for the GPU. For the "scheduler" component, we simply take advantage of the TBB task scheduler to schedule all CPU threads and we dedicate one CPU thread to handle all hand-shaking with the GPU. The "Dev. tools" component is still under development.

In the rest of this section, we will focus our discussion on the Qilin API and the Qilin compiler. We will discuss our adaptive mapping technique in Section 4.

3.1 Qilin API

Qilin defines two new types `QArray` and `QArrayList` using C++ templates. A `QArray` represents a multidimensional array of a generic type. For example, `QArray<float>` is the type for an array of floats. `QArray` is an opaque type and its actual implementation is hidden from the programmer. A `QArrayList` represents a list of `QArray` objects, and is also an opaque type.

Qilin parallel computations are performed on either `QArray`'s or `QArrayList`'s. There are two approaches to writing these computations. The first approach is called the *Stream-API approach*, where the programmer solely uses the Qilin stream API which implements common data-parallel operations as listed in Figure 4. Our stream API is similar to those found in some GPGPU systems [3, 9, 24, 29, 34]. However, since Qilin targets for heterogeneous machines, it also allows the programmer to optionally select the processing elements. For instance, "`QArray<float> Qsum = Add(Qx, Qy, PE_SELECTOR_GPU)`" specifies that the addition of the two arrays `Qx` and `Qy` must be performed on the GPU. Other possible selector values are `PE_SELECTOR_CPU` for choosing the CPU and `PE_SELECTOR_DEFAULT` for choosing the default mapping scheme which can be a static one or the adaptive mapping scheme (See Section 4).

Figure 5 shows a function `qilin_pi()` which uses the Qilin stream API to compute the value of π. The function `QArray::Create2D()` creates a 2D `QArray` from a normal C array. Its reverse is `QArray::ToNormalArray()`, which converts a `QArray` back to a normal C array. The functions `Sqrt`, `*`, `+`, and `<=` are elementwise operations applied to the entire arrays. The function `Sum` performs reduction sum which results into a single value.

The second approach to writing parallel computations is called the *Threading-API approach*, in which the programmer provides the parallel implementations of computations in the threading APIs on which Qilin is built (i.e., TBB on the CPU side and CUDA on the GPU side for our current implementation). A Qilin operation is then defined to glue these two implementations together. When Qilin compiles this operation, it will automatically partition computations across processing elements and eventually merge the results.

Figure 6 shows an example of using the Threading-API approach to write an image filter. First, the programmer provides a TBB implementation of the filter in `CpuFilter()` and a CUDA implementation in `GpuFilter()` (TBB and CUDA codes are omitted for clarity reasons). Since both TBB and CUDA work on normal arrays, we need to convert Qilin arrays back to normal arrays before invoking TBB and CUDA. Second, we use the Qilin function `MakeQArrayOp()` to make a new operation `myFilter` out of `CpuFilter()` and `GpuFilter()`. Third, we construct the argument list of `myFilter` from the two Qilin arrays `qSrc` and `qDst`. The keyword `QILIN_PARTITIONABLE` tells Qilin that the associated computations of both arrays can be partitioned to run on the CPU and GPU. Fourth, we call another Qilin function `ApplyQArrayOp()` to apply `myFilter` with the argument list using the default mapping scheme. The result of `ApplyQArrayOp()` is a Qilin boolean array `qSuccess` of a single element, which returns whether the operation is applied successfully. Finally, we convert `qSuccess` to a normal boolean value.

3.2 Dynamic Compilation

Qilin uses *dynamic* compilation to compile Qilin API calls into native machine codes while the program runs. The main advantage of dynamic over static compilation is being able to *adapt* to changes in the runtime environment. The downside of dynamic compilation is the compilation overhead incurred at run time. However, we argue that this overhead is largely amortized in the typical Qilin usage model, where a relatively small program runs for a long period of time.

The Qilin dynamic compilation consists of the four steps listed below. Figure 7 illustrates these steps during the compilation of the Qilin program shown in Figure 5.

1. **Building Directed Acyclic Graph (DAGs) from Qilin API calls:** DAGs are built according to the *data dependencies* among `QArrays` in the API calls. These DAGs are essentially the intermediate representation which latter steps in the compilation process will operate on.

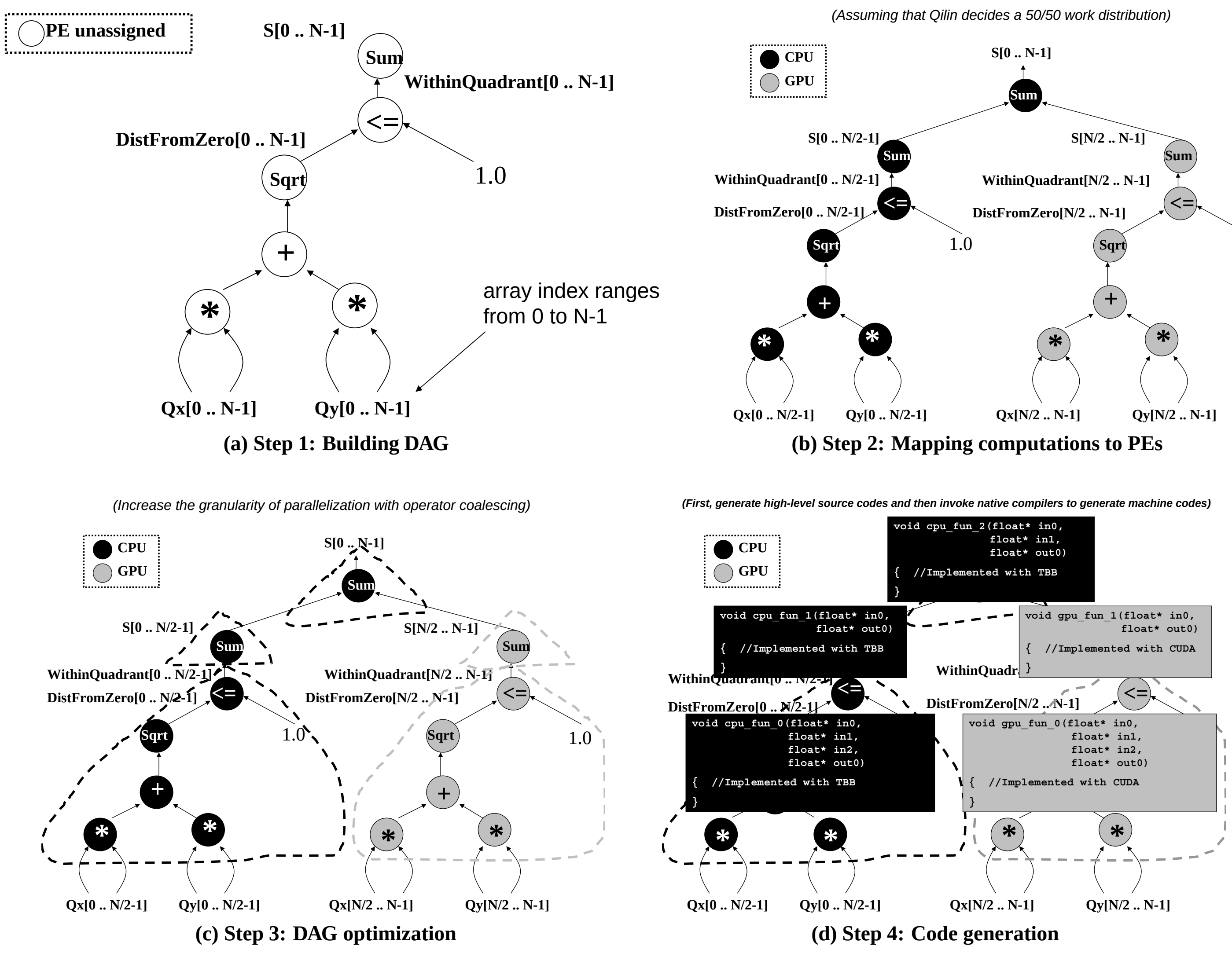
2. **Deciding the mapping from computations to processing elements:** This step uses the automatic adaptive mapping technique to decide the mapping. The details will be given in Section 4.

3. **Performing optimizations on DAGs:** A number of optimizations are applied to the DAGs. The most important ones are (i) operation coalescing and (ii) removal of unnecessary temporary arrays. Operation coalescing groups as many operations running on the same processing elements into a single "super-operation" as possible, and thereby reducing the overhead of scheduling individual operations. In Figure 7(c), for instance, we coalesce elementwise operations into "super-operations" but leave reduction operations alone. It is also important to remove the allocating/deallocating and copying of temporary arrays used in the intermediate computations of `QArrays`.

4. **Code generation:** At this point, we have the optimized DAGs and their computation-to-processor mappings decided. One additional step we need to do here is to ensure that all hardware resource constraints are satisfied. The most common issue is the memory requirement of computations that are mapped to the GPU, because the amount of GPU physical memory available is relatively limited (typically less than 1GB) and it does not have virtual memory. To cope with this issue, Qilin estimates the memory requirement of each DAG scheduled on the GPU. If the memory requirement exceeds the GPU physical memory size, Qilin will automatically split the DAG into multiple smaller DAGs and run them *sequentially*. Once all resource constraints are taken care of, Qilin generates the native machine codes from the DAGs according to the mappings. Qilin also automatically generates all gluing codes needed to combine the results from the CPU and the GPU.

3.2.1 *Reducing Compilation Overhead*

To reduce the runtime overhead, dynamic compilation is mostly done lazily: When a Qilin program is executed, DAGs are built (i.e., Step 1 in the compilation process) as API calls are encountered. Nevertheless, the remaining three steps are not invoked until the computation results are really needed—when we perform a


PE unassigned
S[0 .. N-1]
Sum
WithinQuadrant[0 .. N-1]
<=
DistFromZero[0 .. N-1]
Sqrt
1.0
+
*
array index ranges from 0 to N-1
Qx[0 .. N-1]
Qy[0 .. N-1]
(a) Step 1: Building DAG
(Assuming that Qilin decides a 50/50 work distribution)
CPU
GPU
S[0 .. N/2-1]
S[N/2 .. N-1]
WithinQuadrant[0 .. N/2-1]
WithinQuadrant[N/2 .. N-1]
DistFromZero[0 .. N/2-1]
DistFromZero[N/2 .. N-1]
Qx[0 .. N/2-1]
Qy[0 .. N/2-1]
Qx[N/2 .. N-1]
Qy[N/2 .. N-1]
(b) Step 2: Mapping computations to PEs
(Increase the granularity of parallelization with operator coalescing)
(c) Step 3: DAG optimization
(First, generate high-level source codes and then invoke native compilers to generate machine codes)
(d) Step 4: Code generation

```
void cpu_fun_2(float* in0,
               float* in1,
               float* out0)
{  //Implemented with TBB
}
```

```
void cpu_fun_1(float* in0,
               float* out0)
{  //Implemented with TBB
}
```

```
void gpu_fun_1(float* in0,
               float* out0)
{  //Implemented with CUDA
}
```

```
void cpu_fun_0(float* in0,
               float* in1,
               float* in2,
               float* out0)
{  //Implemented with TBB
}
```

```
void gpu_fun_0(float* in0,
               float* in1,
               float* in2,
               float* out0)
{  //Implemented with CUDA
}
```

Figure 7: Compilation of the Qilin program in Figure 5. Each node represents a computation and each edge represents a data dependence. The DAG in Step 1 is duplicated in Step 2 since Qilin decides to map it to both CPU and GPU. In Step 3, operations within the same dotted lines are grouped into a single "super-operation". In Step 4, a TBB function is generated for each "super-operation" on the CPU and a CUDA function is generated for each "super-operation" on the GPU. Note that the number of input/output parameters of a function is equal to the number of fanins/fanouts of the corresponding "super-operation".

`QArray::ToNormalArray()` call to convert from a `QArray` to a normal C array. Thus, dynamically dead Qilin API calls will not cause any compilation. In addition, code generated for a particular DAG is stored in a software code cache so that if the same DAG is seen again later in the same program run, we do not have to redo Steps 2-4.

4. ADAPTIVE MAPPING

The Qilin adaptive mapping is a technique to automatically find the near-optimal mapping from computations to processing elements for the given application, problem size, and system configuration. To facilitate our discussion, let us introduce the following notations:

$$\begin{aligned}
T_C(N) &= \text{The } \textit{actual} \text{ time to execute the} \\
&\quad\ \text{given program of problem size } N \text{ on the CPU.} \\
T_G(N) &= \text{The } \textit{actual} \text{ time to execute the} \\
&\quad\ \text{given program of problem size } N \text{ on the GPU.} \\
T'_C(N) &= \text{Qilin's projection of } T_C(N) \\
T'_G(N) &= \text{Qilin's projection of } T_G(N)
\end{aligned}$$

One approach to predicting $T_C(N)$ and $T_G(N)$ is to build an *analytical* performance model based on static analysis. While this approach might work well for simple programs, it is unlikely to be sufficient for more complex programs. In particular, predicting $T_C(N)$ using an analytical model is challenging with features like out-of-order execution, speculation and prefetching on modern

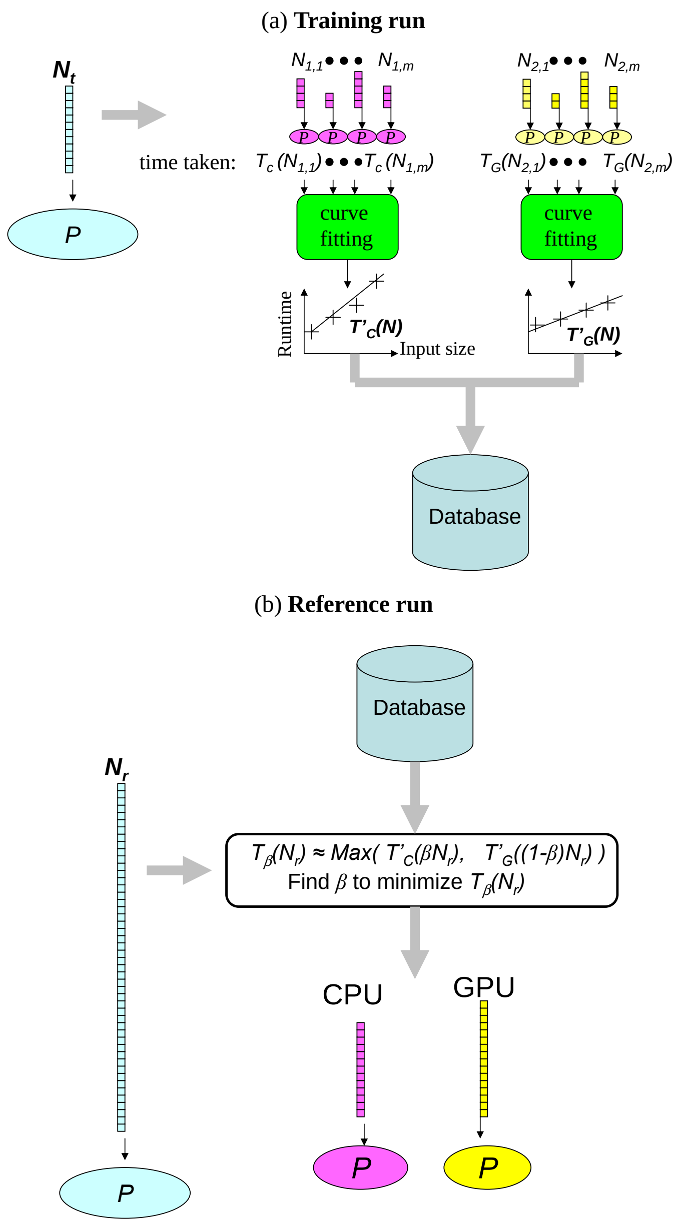

(a) Training run
N_t
$N_{1,1}$ ••• $N_{1,m}$
$N_{2,1}$ ••• $N_{2,m}$
P
time taken: $T_c(N_{1,1})$ ••• $T_c(N_{1,m})$
$T_G(N_{2,1})$ ••• $T_G(N_{2,m})$
curve fitting
curve fitting
Runtime
Input size
$T'_c(N)$
$T'_G(N)$
P
Database
(b) Reference run
Database
N_r
$T_\beta(N_r) \approx Max(T'_C(\beta N_r), T'_G((1-\beta)N_r))$
Find β to minimize $T_\beta(N_r)$
CPU
GPU
P
P
P

Figure 8: Qilin's adaptive mapping technique.

processors. Instead, Qilin takes an *empirical* approach, as depicted in Figure 8 and explained below.

Qilin maintains a database that provides execution-time projection for all the programs it has ever executed. The first time that a program is run under Qilin, it is used as the *training* run (see Figure 8(a)). Suppose the input problem size for this training run is N_t. Then Qilin divides the input into two parts of size N_1 and N_2. The N_1 part is mapped to the CPU while the N_2 part is mapped to the GPU. Within the CPU part, it further divides N_1 into m subparts $N_{1,1}...N_{1,m}$. Each subpart $N_{1,i}$ is run on the CPU and the execution time $T_C(N_{1,i})$ is measured. Similarly, the N_2 part is further divided into m subparts $N_{2,1}...N_{2,m}$ and their execution times on the GPU $T_G(N_{2,i})$'s are measured. Once all $T_C(N_{1,i})$'s and $T_G(N_{2,i})$'s are available, Qilin uses curve fitting to construct two linear equations $T'_C(N)$ and $T'_G(N)$ as projections for the ac-

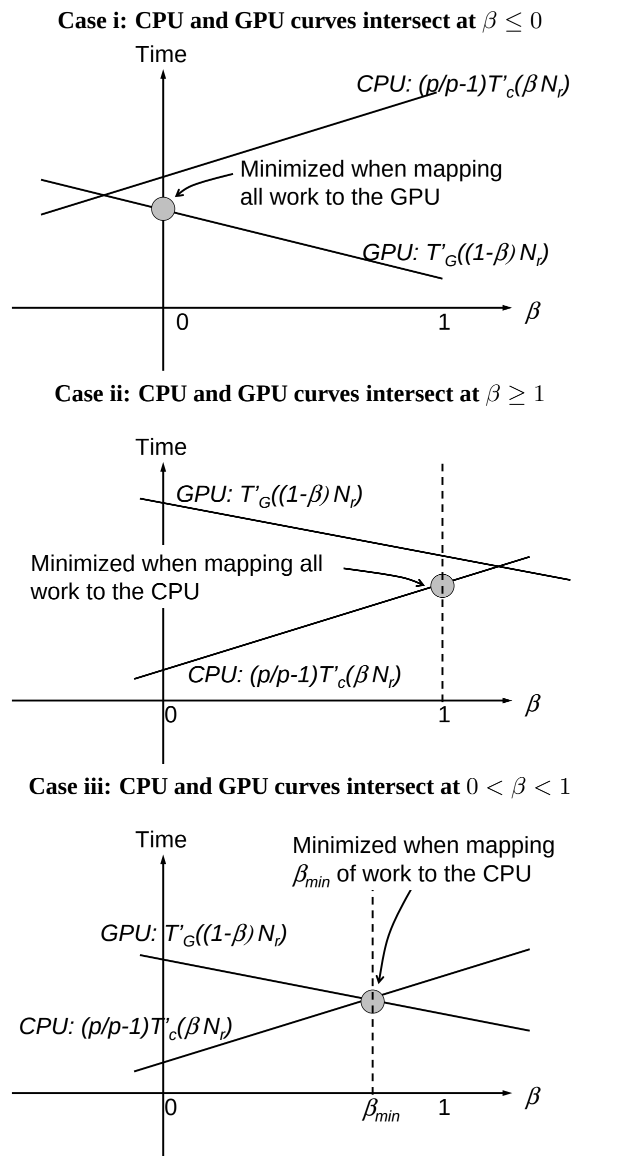


Figure 9: Three possible cases of the β in Equation (4).

tual execution times $T_C(N)$ and $T_G(N)$, respectively. That is:

$$\begin{aligned} T_C(N) &\approx T'_C(N) \\ &= a_c + b_c * N \end{aligned} \quad (1)$$

$$\begin{aligned} T_G(N) &\approx T'_G(N) \\ &= a_g + b_g * N \end{aligned} \quad (2)$$

where a_c, b_c, a_g and b_g are constant real numbers. The next time that Qilin runs the same program with a different input problem size N_r, it can use the execution-time projection stored in the database to determine the computation-to-processor mapping (see Figure 8(b)). Let β be the fraction of work mapped to the CPU and $T_\beta(N)$ be the actual time to execute β of work on the CPU and $(1-\beta)$ of work on the GPU in *parallel*. Then:

$$\begin{aligned} T_\beta(N) &= Max(T_C(\beta N), T_G((1-\beta)N)) \\ &\approx Max(T'_C(\beta N), T'_G((1-\beta)N)) \end{aligned} \quad (3)$$

	CPU	GPU
Architecture	Intel Core2 Quad	Nvidia 8800 GTX
Core Clock	2.4 GHz	575 MHz
Number of Cores	8 cores (on two sockets)	128 stream processors
Memory size	4 GB	768 MB
Memory Bandwidth	8 GB/s	86.4 GB/s
Threading API	Intel Threading Building Blocks (TBB)	Nvidia CUDA 1.1
Compiler	Intel C Compiler; (ICC 10.1, “-fast”)	Nvidia C Compiler (NVCC 1.1,“-O3”)
OS	32-bit Linux Fedora Core 6	

Table 1: Experimental Setup

The above equations assume that running the CPU and GPU simultaneously is as fast as running them standalone. In reality, this is not the case since the CPU and GPU do contend for common resources including bus bandwidth and the need of CPU processor cycles to handshake with the GPU. So, we take these factors into account by multiplying factors into Equation (3):

$$T_\beta(N) \approx Max(\frac{p}{p-1}T'_C(\beta N), T'_G((1-\beta)N)) \quad (4)$$

where p is the number of CPU cores. Here we dedicate one CPU core to handshake with the GPU and we assume that bus bandwidth is not a major factor.

Now, we can plug the input problem size N_r into Equation (4). $T'_C(\beta N_r)$ becomes a linear equation of a single variable β and the same for $T'_G((1-\beta)N_r)$. We need to find the value of β that minimizes $Max(\frac{p}{p-1}T'_C(\beta N_r), T'_G((1-\beta)N_r))$. This is equivalent to finding the β at which $\frac{p}{p-1}T'_C(\beta N_r)$ and $T'_G((1-\beta)N_r)$ intersect. There are three possible cases, as illustrated in Figure 9. In case (i) where the intersection happens at $\beta \leq 0$, Qilin maps all work to the GPU; in case (ii) where the intersection happens at $\beta \geq 1$, Qilin maps all work to the CPU; in case (iii) where the intersection happens at $0 < \beta_{min} < 1$, Qilin maps β_{min} of work to the CPU and $1 - \beta_{min}$ of work to the GPU.

Finally, if Qilin detects any hardware changes since the last training run for a particular program was done, it will trigger a new training run for the new hardware configuration and use the results for future performance projection.

5. EVALUATION

We now present the methodology used to evaluate the effectiveness of adaptive mapping and the results.

5.1 Methodology

Our evaluation was done on a heterogeneous PC, consisting of a multicore CPU and a high-end GPU, as depicted in Table 1. Table 2 lists our benchmarks, which are computation kernels used in different domains including financial modeling, image processing, and scientific computing. They were also used in the Merge work [16], the most relevant previous study. We compare Qilin's adaptive implementation against the best available CPU or GPU implementations whenever possible: for the GPU implementations, we use the ones provided in the CUDA Software Development Kit (SDK) [19] if available.

We measured the wall-clock execution time, including the time required to transfer the data between the CPU and the GPU when the GPU is used. For Qilin's adaptive versions, we did one training run followed by a variable number of reference runs until the total execution time (including both training and reference runs) reaches one hour. We then report the average execution time per reference run (i.e., total execution time divided by the number of reference runs). This emulates the expected usage model of Qilin where a program is trained once and then used many times afterwards. In addition, since our current prototype uses source-to-source translation and invokes the system compilers to generate the final executables (see Section 3), the dynamic compilation overhead is fairly high (about 10 secs). So we amortize it with multiple reference runs. In a production environment where we shall have a full-fledged compiler that generates code all the way from source to binary, the compilation overhead would be much less an issue.

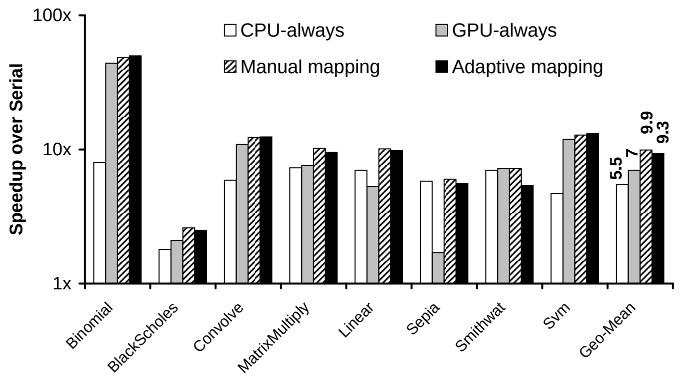


Figure 10: Performance of Adaptive Mapping (Note: The y-axis is in logarithmic scale).

5.2 Results

In this section, we first present results that compare adaptive mapping against manual mapping, using training set sizes identical to the reference set sizes. Second, we present results with different training input sizes. Finally, we present results with different hardware and software configurations.

5.2.1 *Effectiveness of Adaptive Mapping*

Figure 10 shows the performance comparison between automatic adaptive mapping and other mappings. The y-axis is the speedup over the serial case in logarithmic scale. The x-axis shows the benchmarks and the geometric mean. In each benchmark, there are four bars representing different mapping schemes. “CPU-always” means always scheduling all the computations to the CPU, while “GPU-always” means scheduling all the computations to the GPU. “Manual mapping” means the programmer *manually* determines the near-optimal mapping by trying all possible CPU/GPU work distributions at a granularity of 10%. “Adaptive mapping” means Qilin *automatically* determines the near-optimal mapping via adaptive mapping. Both “Manual mapping” and “Adaptive mapping” use training inputs that are identical to the reference inputs (we will investigate the impact of different training inputs in Section 5.2.3).

Table 3 reports the distribution of computations under the last two mapping schemes. Figure 10 shows that, on average, adaptive mapping is 41% faster than always using the CPU and 25% faster than always using the GPU. It is within 94% of the near-optimal mapping found manually. In the cases of `Binomial`, `Convolve` and `Svm`, adaptive mapping performs slightly better than manual mapping because it can distribute works at a finer granularity.

There are two major factors that determine the performance of adaptive mapping. The first factor is how accurate Qilin's performance projections are (i.e., Equation (4) in Section 4). Table 3 shows that the work distributions under the two mapping schemes

Benchmark	**Description**	**Input Problem Size**	**Serial Time**	**Origin**
Binomial	American option pricing	1000 options, 2048 steps	11454 ms	CUDA SDK [19]
BlackScholes	Eurpoean option pricing	10000000 options	343 ms	CUDA SDK
Convolve	2D separable image convolution	12000 x 12000 image, kernel radius = 8	10844 ms	CUDA SDK
MatrixMultiply	Dense matrix multiplication	6000 x 6000 matrix	37583 ms	CUDA SDK
Linear	Linear image filter—compute output pixel as average of a 9-pixel square	13000 x 13000 image	6956 ms	Merge [16]
Sepia	Modify RGB value to artificially age images	13000 x 13000 image	2307 ms	Merge
Smithwat	Compute scoring matrix for a pair of DNA sequences	2000 base pairs	26494 ms	Merge
Svm	Kernel from a SVM-based face classifier	736 x 992 image	491 ms	Merge

Table 2: Benchmarks

	Manual mapping		**Adaptive mapping**	
	CPU	**GPU**	**CPU**	**GPU**
Binomial	10%	90%	10.5%	89.5%
BlackScholes	40%	60%	46.5%	53.5%
Convolve	40%	60%	36.3%	63.7%
MatrixMultiply	40%	60%	45.5%	54.5%
Linear	60%	40%	50.8%	49.2%
Sepia	80%	20%	76.2%	23.8%
Smithwat	60%	40%	59.3%	40.7%
Svm	10%	90%	14.3%	85.7%

Table 3: Distribution of computations under the manual mapping and adaptive mapping in Figure 10. For manual mapping, we use a granularity of 10%.

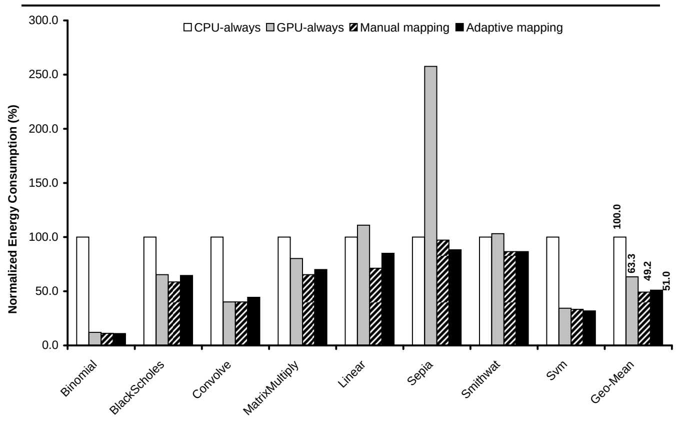


Figure 11: Energy consumption of various mapping schemes (normalized to the CPU-always case).

are similar, indicating that Qilin's performance projections are fairly accurate. The second factor is the runtime overhead of adaptive mapping. Among our benchmarks, `Smithwat` is the only one that significantly suffers from this overhead (see the relative low performance of "Adaptive mapping"for `Smithwat` in Figure 10). This benchmark computes the scoring matrix for a pair of DNA sequences. The serial version consists of a two-level for-loop which considers all possible pairs of elements. Data dependency constrains us to parallelize the inner loop instead of the outer loop. Consequently, the Qilin version has to pay the adaptation overhead in each iteration of the outer loop.

5.2.2 *Energy Consumption*

We also studied the energy consumption under various mapping schemes. We measured the entire system power (including the CPU, GPU, memory, and disks) using Extech 380801 Power Analyzer [6] and obtained the total energy consumption by computing the product of power and execution time. Figure 11 shows our measurements. For "Manual mapping", the programmer searches for the distributions that result into minimal energy (again at the granularity of 10%). For "Adaptive mapping", Qilin's adaptation is still based on *execution time* but the energy consumption is reported. The relatively high energy consumption of the GPU in the `Sepia` case is a direct result of the GPU's sub-optimal performance in this benchmark. The `Sepia` kernel has three conditional branches in computing each pixel and this could be why the GPU performance suffers. Overall, the result in this section shows that adaptive mapping is also effective in terms of energy: it consumes 49% less energy than CPU-only and 20% less than GPU-only, and is within 96% of the energy efficiency of manual mapping.

5.2.3 *Impact of Training Input Size*

Figure 12 shows the impact of the training set size on the performance of adaptive mapping. Each benchmark uses six different training set sizes, ranging from 10% to 100% of the reference set size (i.e., the "100%" bars in Figure 12 are the same as the "Adaptive mapping" bars in Figure 10). Figure 12 shows that much of the performance benefit of adaptive mapping is preserved as long as the training set size is at least 30% of the reference set size. When the training set size is only one-tenth of the reference set size, the average adaptive-mapping speedup drops to 7.5x, but is still higher than the 7.0x or 5.5x speedups by using the GPU or CPU alone, respectively. These results provide a guideline on when Qilin should apply adaptive mapping if the actual problem sizes are significantly different from the training input sizes stored in the database.

5.2.4 *Adapting to Hardware Changes*

One important advantage of adaptive mapping is its capability to adjust to hardware changes. To demonstrate this advantage, we did the following two experiments.

In the first experiment, we replaced our GPU by a less powerful one (Nvidia 8800 GTS) but kept the same 8-core CPU. 8800 GTS has fewer stream processors (96) and less memory (640MB)

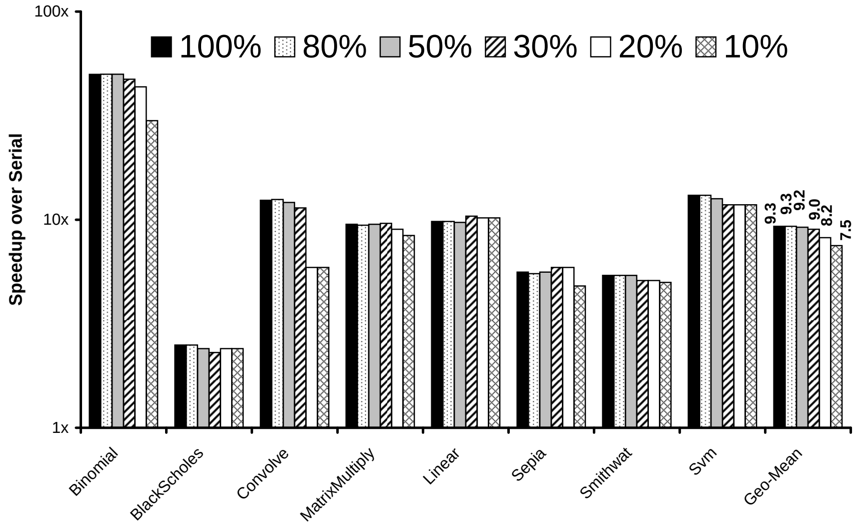


Figure 12: Impact of the training set size on adaptive mapping performance (Note: The y-axis is in logarithmic scale. The legend ”X%” means the training set size is X% of the reference set size.).

than 8800 GTX. The performance of adaptive mapping with this new hardware configuration is shown in Figure 13(a). The “GPU-always” speedup with 8800 GTS is 5.7x compared to the 7x speedup with 8800 GTX in Figure 10, or a 19% performance reduction. Adaptive mapping automatically re-distributes the computations for this change. Consequently, adaptive mapping achieves a 8.2x speedup, or a 12% performance reduction compared against its 9.3x speedup in Figure 10. Note that the performance reduction in the “Adaptive mapping” case (12%) is less than that in the “GPU-always” case (19%), because Qilin is able to recover part of the loss from the CPU.

In the second experiment, we went for the other extreme, where we replaced our 8-core CPU by a 2-core CPU but kept the original GPU (8800 GTX). The performance with this new hardware configuration is shown in Figure 13(b). With a 2-core CPU, the average speedup of “CPU-always” is down to 1.5x. Qilin decided to shift most computations to the GPU. As a result, the “Adaptive mapping” speedup in Figure 13(b) is only slightly better than the “GPU-always” speedup.

5.2.5 Adapting to Software Changes

Adaptive mapping can also adjust to software changes. In this experiment, we changed the CPU compiler from ICC to GCC. GCC generally generates less optimized code than ICC, especially in SIMDization. As shown in Figure 14, both the baseline serial performance and the CPU-always parallel performance drop significantly. Fortunately, adaptive mapping recovers much of this performance loss by shifting more work to the GPU.

6. RELATED WORK

Heterogeneous multiprocessors have been drawing increasing attentions from both the hardware and software research communities. On the hardware side, Kumar *et al.* [14] demonstrate the advantages of heterogeneous chip multiprocessors (CMPs) over homogeneous CMPs in terms of power and throughput. They predict that once homogeneous CMPs reach a total of four cores, the benefits of heterogeneity will outweigh the benefits of additional homogeneous cores in many applications. They also classify heterogeneous multiprocessors into multi-ISA multiprocessors such as

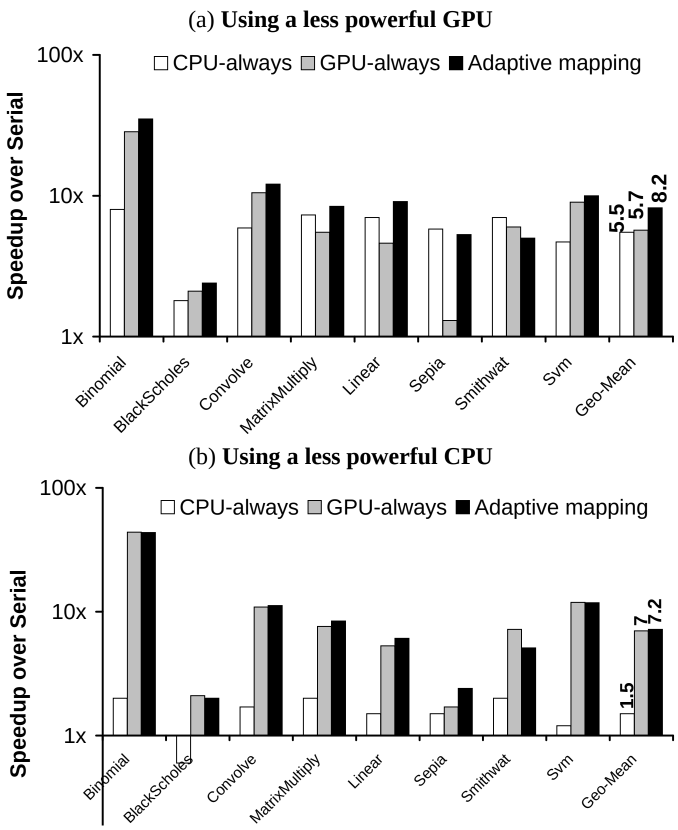


Figure 13: Performance of Adaptive Mapping with respect to hardware change (Note: The y-axis is in logarithmic scale).

the Cell and CPU+GPU and single-ISA multiprocessors [13] such as the upcoming Intel’s Larrabee [32]. Our adaptive mapping technique is applicable to both single-ISA or multi-ISA heterogeneous multicores. Recently, Hill and Marty [10] also argue that hetero-

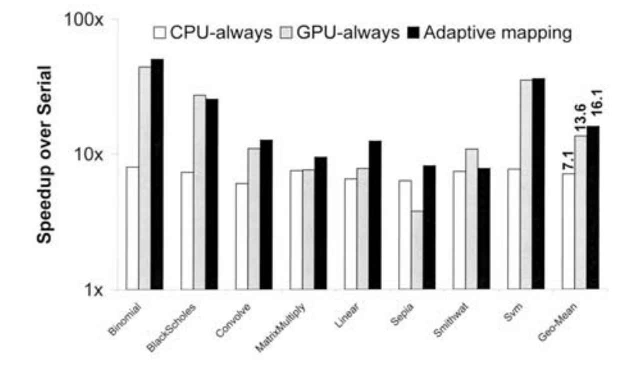


Figure 14: Performance of Adaptive Mapping when the GCC compiler is used on the CPU (Note: The y-axis is in logarithmic scale).

geneous ("asymmetric" in their terminology) multicore designs offer greater potential speedup than homogeneous ("symmetric") designs, provided that software challenges like the computation-to-processor mapping problem can be addressed.

On the software side, there are a number of GPGPU programming systems from both academic and industry. Table 4 compares Qilin with these systems. Many of them, including Stanford's Brook [3], Microsoft's Accelerator [34], Google's Peakstream [24], AMD's Brook+ [1] and Nvidia's Cuda [21] target for only the GPU. In contrast, Intel's Ct [9] currently targets for the CPU only. Liao *et al.* [15] developed a compiler that generates multithreaded CPU codes from Brook-like programs. Similarly, Stratton *et al.* [33] have developed MCUDA for translating CUDA kernels to run on multicore CPUs. RapidMind [29], Cell [26, 5, 22], Merge [16], and Qilin target both CPU and GPU. The key advantage of Qilin is that the mapping is done automatically and is adaptive to changes in the runtime environment. Most recently, OpenCL [18] is proposed as the standard API for programming GPUs. At this moment, it is unclear what kind of mechanism will be available in OpenCL to help programmers decide the computation-to-processor mapping.

Recently, Jimenez *et al.* [12] propose a user-level scheduler based on past performance history for heterogeneous systems. Their goal is to speed up the scenario where *multiple* applications are run simultaneously on the system. In contrast, we focus on improving the performance and energy consumption of a *single* application. So, unlike our work, they do not distribute the computation within a single application. Because of these differences in the focus, they use the runtime scheduler approach while we use the dynamic compilation approach. In addition, our dynamic compiler also performs code generation for the CPU and GPU (see Section 3.2). At the operating system level, Ghiasi *et al.* [8] propose a scheduler that schedules memory-bound tasks to cores running at lower frequencies, thereby limiting system power while minimizing total performance loss.

Profiling is a proven technique in static [25, 17] and dynamic compilation [2]. Most recently, Wang and O'Boyle [37] uses *offline* profiling to build a machine learning based model which predicts the optimal number of OpenMP threads and scheduling policies on a homogeneous architecture. In contrast, our work is based on the *online* profiling done during dynamic compilation and our goal is to find the optimal work distribution between the CPU and GPU.

Our work is also related to program *autotuning* [4, 7, 23, 27, 28, 31, 35, 38], an increasingly popular approach to producing high-quality portable code by generating many variants of a computation kernel and benchmarking each variant on the target platform. Existing autotuners largely focus on tuning the program parameters that affect the memory-hierarchy performance, such as the cache blocking factor and prefetch distance. Adaptive mapping can be viewed as an autotuning technique that tunes for the distribution of works on heterogeneous multiprocessors.

7. CONCLUSION

We have presented adaptive mapping, an automatic technique to map computations to processing elements on heterogeneous multiprocessors. We have implemented it in an experimental system called Qilin for programming CPUs and GPUs. We demonstrate that automated adaptive mapping performs close to manual mapping in both execution time and energy consumption and can adapt to changes in input problem sizes, hardware and software configurations. We believe that adaptive mapping could be an important technique in the multicore software stack.

8. ACKNOWLEDGMENTS

We thank Michael Linderman, Jamison Collins, and Hong Wang for providing the Merge benchmarks; Geoff Lowney and Mark Abel for supporting this work; Robert Cohn, Richard Vuduc, and Nagesh B. Lakshminarayana for suggestions and comments. Sunpyo Hong and Hyesoon Kim are partially supported by NSF CCF0903447, NSF/SRC task 1981, Microsoft Research, Intel, equipment donations from NVIDIA, and power measurement equipment from CERCS.

9. REFERENCES

[1] AMD. *AMD Stream SDK User Guide v 1.2.1-beta*, Oct 2008.

[2] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. Adaptive Optimization in the Jalapeno JVM. In *Proceedins of OOPSLA'00* (October 2000).

[3] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics 23*, 3 (2004), 777–786.

[4] CHEN, C., CHAME, J., NELSON, Y. L., DINIZ, P., HALL, M., AND LUCAS, R. Compiler-Assisted Performance Tuning. In *Proceedings of SciDAC 2007, Journal of Physics: Conference Series* (June 2007).

[5] EICHENBERGER, A. E., O'BRIEN, K., O'BRIEN, K., WU, P., CHEN, T., ODEN, P. H., PRENER, D. A., SHEPHERD, J. C., SO, B., SURA, Z., WANG, A., ZHANG, T., ZHAO, P., AND GSCHWIND, M. Optimizing Compiler for a CELL Processor. In *Proceedings of the 2005 International Conference on PACT*.

[6] EXTECH. *Extech Power Analyzer 380801*. http://www.extech.com.

[7] FURSIN, G. G., O'BOYLE, M. F. P., AND KNIJNENBURG, P. M. W. Evaluating Iterative Compilation. In *Proceedings of the 2002 Workshop on Languages and Compilers for Parallel Computing*.

[8] GHIASI, S., KELLER, T., AND RAWSON, F. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings of the 2nd Conference on Computing Frontiers* (May 2005), pp. 199–210.

[9] GHULOUM, A., SMITH, T., WU, G., ZHOU, X., FANG, J., GUO, P., SO, B., RAJAGOPALAN, M., CHEN, Y., AND CHEN, B. Future-Proof Data Parallel Algorithms and Software On Intel Multi-Core Architecture. *Intel Technology Journal 11*, 4, 333–348.

[10] HILL, M., AND MARTY, M. R. Amdahl's Law in the Multicore Era. *IEEE Computer* (July 2008), 33–38.

[11] INTEL. *Intel Math Kernel Library Reference Manual*, Sept 2007.

	Brook [3]	PeakStream [24]	RapidMind [29]	MS Accelerator [34]	Cell [26]	Ct [9]	Cuda [21]	MCUDA [33]	Merge [16]	Qilin
Programming Model	S-API	S-API	S-API	S-API	S-API + C-API	S-API	C-API	C-API	C-API	S-API + C-API
Hardware	GPU	GPU	CPU+GPU	GPU	CPU+SPU	CPU	GPU	CPU	CPU+GPU	CPU+GPU
Compilation	Dynamic	Dynamic	Dynamic	Dynamic	Static	Dynamic	Static	Static	Static	Dynamic
Mapping to Processing Elements	Fixed	Fixed	Manual	Fixed	Manual	Fixed	Fixed	Fixed	Manual	Automatic

Table 4: A comparison of Qilin with other similar systems. ("S-API" = Stream API; "C-API" = C-based API. "Fixed" mapping means computations are mapped to either CPU or GPU but not both. "Manual" or "Automatic" mapping means that computations are manually or automatically distributed over the CPU and GPU, respectively.)

[12] JIMENEZ, V. J., VILANOVA, L., GELADO, I., GIL, M., FURSIN, G., AND NAVARRO, N. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers* (2009), pp. 19–33.

[13] KUMAR, R., FARKAS, K. I., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. Single-ISA Heterogeneous Multicore Architectures: The Potential for Processor Power Reduction. In *Proceedings of the MICRO'03* (December 2003), pp. 81–92.

[14] KUMAR, R., TULLSEN, D., JOUPPI, N., AND RANGANATHAN, P. Heterogeneous Chip Multiprocessors. *IEEE Computer* (November 2005), 32–38.

[15] LIAO, S.-W., DU, Z., WU, G., AND LUEH, G.-Y. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the 4th Conference on CGO* (March 2006), pp. 196–207.

[16] LINDERMAN, M. D., COLLINS, J. D., WANG, H., AND MENG, T. H. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proceedings of the 2008 ASPLOS* (March 2008).

[17] LUK, C.-K., MUTH, R., PATIL, H., COHN, R., AND LOWNEY, P. G. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of 2004 CGO* (2004), pp. 15–26.

[18] MUNSHI, A. OpenCL Parallel Computing on the GPU and CPU. In *ACM SIGGRAPH 2008* (2008).

[19] NVIDIA. *CUDA SDK*. http://www.nvidia.com/object/cuda_get.html.

[20] NVIDIA. *CUDA CUBLAS Reference Manual*, June 2007.

[21] NVIDIA. *CUDA Programming Guide v 1.0*, June 2007.

[22] O'BRIEN, K., O'BRIEN, K., SURA, Z., CHEN, T., AND ZHANG, T. Supporting OpenMP on Cell. *International Journal on Parallel Programming 36* (2008), 289–311.

[23] PAN, Z., AND EIGENMANN, R. PEAL—A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration. *ACM Transactions. on Programming Languages and Systems 30*, 3 (May 2008).

[24] PEAKSTREAM. *Peakstream Stream Platform API C++ Programming Guide v 1.0*, May 2007.

[25] PETTIS, K., AND HANSEN, R. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 90 Conference on PLDI* (June 1990), pp. 16–27.

[26] PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. M., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., AND YAZAWA, K. The Design and Implementation of a First-Generation CELL Processor. In *IEEE International Solid-State Circuits Conference* (May 2005), pp. 49–52.

[27] POUCHET, L.-N., BASTOUL, C., COHEN, A., AND CAVAZOS, J. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the ACM SIGPLAN 08 Conference on PLDI* (June 2008).

[28] PUSCHEL, M., MOURA, J., JOHNSON, J., PAUDA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GACIC, A., VORONENKO, Y., CHEN, K., JOHNSON, R., AND RIZZOLO, N. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaption 93*, 2 (2005), 232–275.

[29] RAPIDMIND. *Rapidmind*. http://www.rapidmind.net.

[30] REINDERS, J. *Intel Threading Building Blocks*. O'Reilly, July 2007.

[31] REN, M., PARK, J., HOUSTON, M., AIKEN, A., AND DALLY, W. J. A Tuning Framework for Software-Managed Memory Hierarchies. In *Proceedings of the 2008 International Conference on PACT*.

[32] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *Proceedings of ACM SIGGRAPH 2008* (2008).

[33] STRATTON, J. A., STONE, S. S., AND M W. HWU, W. MCUDA: An Efficient Implementation of CUDA Kernels from Multi-Core CPUs. In *Proceedings of the 2008 Workshop on Languages and Compilers for Parallel Computing*.

[34] TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the 2006 ASPLOS* (October 2006).

[35] VUDUC, R., DEMMEL, J., AND YELICK, K. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series* (June 2005).

[36] WANG, P., COLLINS, J. D., CHINYA, G., JIANG, H., TIAN, X., GIRKAR, M., YANG, N., LUEH, G.-Y., AND WANG, H. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the ACM SIGPLAN 07 Conference on PLDI* (June 2007), pp. 156–166.

[37] WANG, Z., AND O'BOYLE, M. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *Proceedings of 2009 ACM PPoPP* (2009), pp. 75–84.

[38] WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing 27*, 1-2 (2001), 3–35.