Analytical Performance Prediction for Evaluation and Tuning of GPGPU Applications

Sara S. Baghsorkhi
University of Illinois at Urbana-Champaign
bsadeghi@illinois.edu

Matthieu Delahaye
University of Illinois at Urbana-Champaign
matthieu@crhc.illinois.edu

William D. Gropp
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

Wen-mei W. Hwu
University of Illinois at Urbana-Champaign
hwu@crhc.illinois.edu

Abstract

In this paper we present an analytical model to predict the performance of general purpose applications on a GPU architecture. The model is designed to provide performance information to an auto-tuning compiler and assist it narrow the search to the more promising implementations. This work is based on the NVIDIA GPUs using CUDA (Compute Unified Device Architecture). We analyze each CUDA kernel and generate the corresponding *string model* which is a concise representation of the operations of a kernel. String model for a kernel summarizes how the kernel exercises major GPU microarchitecture features. Based on the string model we estimate the average execution time of a *warp*, which is the SIMD work granularity for CUDA. We validated the performance model using a few data parallel benchmarks that exploit different microarchitecture features of GPU architecture. The model captures full system complexity and shows high accuracy in predicting the performance trend of different optimized implementations. We also describe our approach to extract the performance model automatically.

Keywords GPGPU, performance analysis

1. Introduction

Graphics processors traditionally had highly specialized programming models and interfaces that limit the ability of developers to map general-purpose applications to these platforms (Owens et al. 2005). With the introduction of NVIDIA's Compute Uniform Device Architecture (CUDA) (NVIDIA 2007) and CUDA-enabled GPUs, developers now have the programming and architectural features to quickly port programs to a platform with a massively-parallel, GPU-based co-processor (Nickolls et al. 2008). The intent of our work is to model the GPU organization and features for analyzing the performance of general purpose applications. In this paper we focus on CUDA-enabled NVIDIA GPUs. However, the core techniques and concepts can be applied to other GPU architectures as well.

1.1 Motivation

The amount of effort required to maximize the performance of programs on GPU architectures can be relatively high. Due to specific resource restrictions and threading model of the GPU the optimization space can also be discontinuous. A study by (Ryoo et al. 2008) demonstrated very large configuration space even for relatively small kernels. The results also concluded that the difference in performance between some manually-optimized variants of code and the optimal configuration was 17%. Furthermore, upgrades to the hardware requires the reapplication of the optimizations.

Empirical performance tuning (Qasem et al. 2004; Whaley 2005; Pan and Eigenmann 2006b) is a well-known technique for solving the above pitfalls. There has also been significant amount of research to develop models and frameworks for predicting performance of applications (Karkhanis and Smith 2004; Marin and Mellor-Crummey 2004; Zhong et al. 2003; Clement and Quinn 1993; Triantafyllis et al. 2003; Pan and Eigenmann 2006a), but it mostly applies to non-GPU architecture.

In this paper we have developed a performance model to help prune the search space of GPU kernel optimizations. Our goal is to use our model as a supporting module for an automated optimizing compiler for GPU architecture. We also intend to export useful programmer's insights about program such as constraints on parameter values, to the performance model. Figure 1 displays the layout of different modules involved in such a compiler framework.

1.2 Performance Factors

GPUs supports the Single-Program Multiple-Data (SPMD) model. Threads within certain granularities (*thread blocks*), share their data and synchronize their actions. During execution threads within a block are grouped into warps (32 parallel threads for the current NVIDIA GPUs), which are the granular multi-threading scheduling units[1]. Threads in a warp are executed in SIMD mode and warps can be interleaved with hardware multi-threading to tolerate intra-warp

[1] Threads in a thread block are numbered along the x direction first and gathered sequentially into warps.

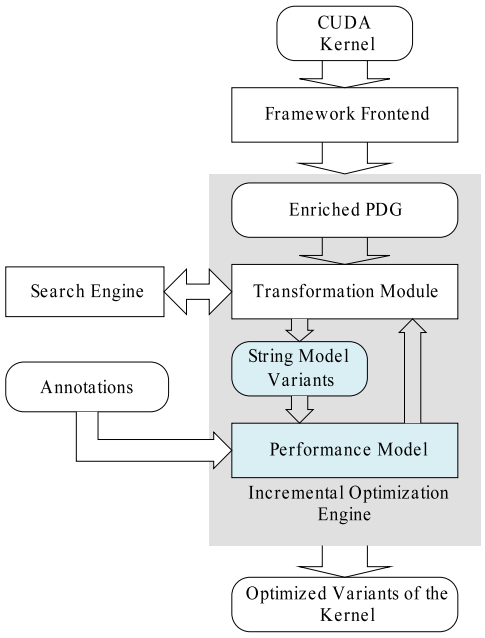


Figure 1. A Compiler Framework Based on Model-guided Empirical Tuning

stalls which enables overlap of memory latency with useful computation. In this section we briefly review major microarchitecture features that are considered for analyzing the performance of a kernel.

1. The general philosophy of GPUs for tolerating memory latency is to generate and maintain thousands of threads. This is in contrast with the use of large caches to hide memory latencies in CPU designs. A high compute-to-memory-access ratio is necessary to avoid saturation of memory channels.
2. To conserve global memory bandwidth, when the threads of a half-warp execute a global load, the loads are consolidated if they meet constraints necessary for the hardware to perform memory coalescing. This is similar to loading an entire cache line from memory versus loading one word at a time.
3. Working memory within a group of cores consists of software-managed cache memory called shared memory. These are high fan-out, low latency, limited-capacity memories which are partitioned among blocks of threads.
4. Shared memories have a limited number of ports, so appropriate thread ordering helps preserve performance by avoiding port and bank conflicts.
5. CUDA is based on the SPMD model in general and the SIMD mode among subsets of threads. Although this is a cost-effective hardware model for exploiting data parallelism, it can be ineffective for algorithms that require diverging control flow decisions in data-parallel sections.

1.3 Contributions

Previous studies on performance estimation (Liu et al. 2007; Govindaraju et al. 2006; Fatahalian et al. 2004) and tuning (Jiang and Snir 2005) for GPU were constrained by the programming environment and the necessity of mapping algorithms to existing GPU features. A more recent work on tuning (Ryoo et al. 2008) uses Pareto-optimal curves to prune optimization space. It introduces efficiency (a flat instruction count) and utilization (a measure of how much of a warp execution time is spent to do useful work compared to the wait time for long latency memory operations) metrics. The proposed performance model in this work captures performance effects of all major GPU microarchitecture features.

We revisited the program dependence graph (PDG) (Ferrante et al. 1987), an intermediate program representation, for the purpose of performance evaluation. The PDG provides a coherent framework to explicitly represent control and data dependences for each program operation. Based on the PDG representation, we can identify computationally related operations in the program that exercise key performance factors. We also explain how to perform symbolic evaluation on these fragments of code efficiently. The combination of a proper program representation and efficient symbolic evaluation enables a compiler framework to capture performance impact of control flow divergence and certain features of memory hierarchy system.

Another major difference between this work and other related work is that we measure each performance factor in isolation and later combine them to model the overall performance. As a result our model properly reflects the interaction between different performance factors.

2. The String Model

To estimate the performance we define an abstract model of computation (we call it $string\ model$) for each GPU kernel. The string model of a kernel is a sequence of the tokens that are shown in Table 1. Each token represents the operational semantics of an instruction or set of instructions along with the expected cost of its execution in the granularity of a warp.

Token C_i^n represents a set of i non-blocking computational instructions. It also infers that on average n clock cycles are required to execute all instances of that block of instructions in a warp. Current NVIDIA GPUs group eight streaming processors into a streaming multiprocessor (SM) that executes the 32 SIMD instruction in a warp by clocking the streaming processors four times. Therefore, we assume that the cost of issuing 32 SIMD instruction in a warp is four clock cycles [2]. If C_i^n is not protected by any condition, it is guaranteed to be executed by all warps of a kernel. Assuming enough warps are scheduled on a streaming multiprocessor to hide pipeline stalls we have $n = 4 \times i$, otherwise

[2] Such implementation dependent information can be provided to the performance model through a hardware specification file.

the pipeline latency that is not tolerated should be counted towards n. If C_i^n is conditionally executed by a subset of threads, n is adjusted according to the fraction of warps for which the guarding condition is satisfied.

M_k^n is the long latency memory read k (a global memory access). The cost for the long latency memory operations of a kernel is calculated by Equation 1, based on the difference between the average number of cycles required to load all the data from global memory (average memory cycles) and the average number of cycles required to perform all non-memory computation (average compute cycles) in a thread block. In Equation 1, NUM_{mem} stands for average number of global memory operations in a thread block; CYC_{mem} and $CYC_{compute}$ are average memory and compute cycles respectively.[3] When CYC_{mem} is less than $CYC_{compute}$, if execution of different warps are interleaved, memory bandwidth is not the critical limiting factor. In that case, n would be the the number of cycles required to issue 32 memory operations in a warp. Otherwise, the cost is adjusted according to the second term in equation 1 to compensate for the number of memory cycles that are not covered by the compute cycles of the kernel.

$$n = \max(4, \frac{CYC_{mem} - CYC_{compute}}{NUM_{mem}}) \qquad (1)$$

Unlike other string tokens, a G_k token does not represent an actual instruction. It indicates that at this point threads in a warp are potentially blocked due to a data dependency to the memory load operation k. To estimate how much of the global memory latency of memory operation k is already covered, we introduce two kernel specific parameters N_{warp} and NBC_{avg}. Number of active warps, N_{warp}, is the maximum number of warps that can be assigned to each streaming multiprocessor in GPU without violating local resource usage (shared memory, registers and other GPU hardware limits). To compute the average number of non-blocking cycles for a warp, NBC_{avg}, we sum up the cost of all string tokens in string model of a kernel, excluding long latency memory tokens M and $\bar{M}$. The result is divided by the dynamic number of blocking tokens (Ms and Bs). On average each of the N_{warp} active warps can use NBC_{avg} cycles before execution is back to a warp that has issued a long latency memory operation. If the memory latency cycles (assumed to be 250 cycles for NVIDIA GPUs) is not totally covered by $(N_{warp} - 1) \times NBC_{avg}$, the non-tolerated latency is included in calculating the average latency of a warp.

String tokens S^n and $\bar{S}^n$ correspond to shared memory load and store operations. Cost n includes the average number of cycles that is required to resolve potential shared memory bank conflicts. Token B indicates a synchronization point in the kernel with a fixed instruction issue cost of 4 clock cycles.

3. Constructing the String Model

Our compiler front-end analyzes the source code and translates it into a program dependence graph (PDG) representation. We perform traditional scalar analysis and optimizations such as induction variable detection and substitution based on an SSA(Cytron et al. 1991) form on the PDG. As a result, program expressions are represented by symbolic expressions in terms of thread ID, block ID and other induction variables. Our framework currently provides closed form expressions for linear and geometric inductive variables. More complex classification of expressions can be supported in the future (Gerlek et al. 1995; Wolfe 1992).

Our framework is also capable of ignoring redundant computation[4] using a simple version of partial evaluation for statements and *value numbering* (Marten Kongstad 2004). Other information such as maximum number of registers required by each thread or the amount of shared memory used by a thread block can also be inferred automatically but it is not supported at this time. We currently rely on NVIDIA's compiler for this information.

Figure 2 shows the program dependence graph after applying scalar optimization for the partial kernel code of prefix sum scan that is shown below.[5]

```
n = 2 * threadID + 1;
/*Load data into shared memory*/
...
for(stride = 2; stride <= 256; stride =<< 1){
  if( ((n+1) % stride  == 0)
      shared[n]+=shared[n - stride >> 1];
  __syncthreads();
}
```

The code is specialized for thread block size of 128 for simplification of our discussion. Two shared memory reads and a shared memory write in each iteration of the loop are under the control of the condition $(n + 1) \equiv 0 \pmod{stride}$.

Rectangular nodes in Figure 2 represent statements in the program. Conditions (predicates) that control execution of a set of instructions (their descendent nodes) have diamond shape. Region nodes, with an oval shape, summarize the set of control condition for a node or set of nodes (Ferrante et al. 1987). Regions also summarizes information such as the average execution weight of the their descendent nodes (usually set to the default value of 1). For example, instructions in a loop should be weighted proportional to the trip count of the loop and therefore each loop region is augmented with the loop trip count,e.g., $W = 8$ in the PDG example of Fig-

[3] Some of the memory operations and computational instructions in a kernel may only be executed by a subset of threads. As a result, CYC_{mem} and $CYC_{compute}$ are average number of cycles instead of total number of cycles.

[4] These are computations that are eventually eliminated by the back-end optimizing compiler.

[5] We simplified and grouped some of the PDG nodes for the ease of explanation.

String Token	Description	Steps to Compute Execution Time of A Warp
C_i^n	A block of i non-blocking continuous computational instructions that would take at least n clock cycles to execute.	Clock+=n
$\bar{M}^n$	A long latency memory write instruction with an associated cost of n cycles due to memory bandwidth limitation.	Clock+=n
M_k^n	The long latency memory load instruction k (250+ cycles) with an associated cost of n cycles due to memory bandwidth limitation.	MemDone[k]=Clock+250, MemStart[k]=Clock, Clock+=n
G_k	A pseudo-instruction which has a data dependency on the earlier long latency memory load operation k.	CoveredLatency = (N_{warps}-1)×NBC_{avg} + (Clock-MemStart[k]) MemWait[k]=max(0,MemDone[k]-Clock-CoveredLatency) Clock+=MemWait[k]
$S^n, \bar{S}^n$	The low latency shared memory load/store instruction with a cost of n cycles. Potential bank conflict penalty is included in n.	Clock+=n
B	A synchronization point in the program.	Clock+=4

Table 1. Algorithm to Estimate Per-Warp Latency

ure 2. This implies that each warp executes the instructions under the loop region 8 times.

In case of a branch divergence, proper weight should be assigned to the two descendent region nodes of the predicate that controls the divergence. For example, the weight that is assigned to the *True* descendent region of a predicate node indicates the fraction of warps that satisfy the condition and therefore execute the instructions under the *True* region.

Divergence can occur at the level of thread blocks, i.e., different thread blocks take different execution paths but all warps in a block still follow the same path. If the condition is expressed in terms of affine expression of block ID, an operator (comparison or modulo), and a constant term, our framework calculates the precise weight for the two descendent regions of the predicate node. Otherwise, our tool assigns weight of zero to both descendent regions of the predicate node to estimate an upper bound and in another pass it sets both weights to 1 to estimate a lower bound for the performance. We intend to integrate the programmer's insight into our tool ,in form of annotations, to generate tight bounds on the performance.

Divergence may also happen at a lower level among warps in a thread block. For example, in the partial prefix sum scan kernel only subsets of the threads in a block are active during each step of execution. Furthermore, these threads are distributed throughout all warps of the thread block for most steps. Therefore, the number of cycles that is required to resolve shared memory bank conflicts is different for each step (If a shared memory access is not dependent on a condition that includes thread ID in x direction, serialization effect can be easily determined by checking the coefficient of the thread ID in the x direction).

In a case similar to the partial kernel code shown above, we symbolically evaluate the conditions and array index expressions that are thread ID dependent. Knowing which threads are active during a step of computation, we determine the number of active warps and pattern of shared memory accesses for each step. Based on this information we compute the average weight for computational instructions (cc) and the average bank conflict penalty for shared memory accesses (cs).[6]

Divergence within a thread block can also make the delays introduced by register dependencies blatant. These delays can be ignored if there are enough warps to hide the intra-warp pipeline latencies. Although a kernel may have enough active warps initially, some of these warps can be turned off for steps with sparse computation pattern. As we determine the number of active warps during symbolic evaluation of a fragment of code, we also estimate the available instruction level parallelism (ILP) within a thread (based on the data dependencies expressed in corresponding regions of PDG). Based on this information we compute the exposed latency cycles for each step. The average cost for computational instructions (cr) is a weighted average of latencies for all steps.

3.1 Efficient Symbolic Evaluation

In this section we propose an efficient approach to symbolic evaluation of conditions in the program that have the triplet form of $\langle At + B,\ op,\ g(i)\rangle$ (similar approach can be adopted to evaluate array index expressions). For brevity, we consider one dimensional thread blocks with the size of T. Due to hardware limitations, T is restricted to 512 (threads). Variable t stands for the thread ID in the affine expression $At + B$. Operator op can be a comparison or modulo operator. Special patterns of bitwise operations can also be translated into a modulo operation, e.g., expressions $t\&(2^i - 1) = 0$ and $t \equiv 0 \pmod{2^i}$ are equivalent.

Function $g(i)$ is a linear ($g(i) = Ci + D$) or geometric ($g(i) = CD^i$) function of the induction variable $i \in \{1, 2, ..., I\}$[7]. Without loss of generality, we restricted our

[6] A similar approach is used to compute *memory cycles* in case of global memory accesses being guarded by conditions on thread ID in x direction.

[7] It is important to note that I can be an arbitrary large number.

discussions to the comparison operator $\leq$ and $A, B, C, D \in \mathbb{N}$. We also assume $g(i) = Ci + D$. Conclusions about the case of $g(i) = CD^i$ can be derived similarly.

The following three mutually exclusive cases apply based on the relationship between T and I and the operator op.

1. $I \leq T$ and $\langle At + B, \leq, g(i)\rangle$.

   We solve $g(i)$ for each $i \in \{1, 2, ..., I\}$. Let $g(i') = Ci' + D$ for $0 \leq i' \leq I$. We have $At + B \leq Ci' + D$ that gives $t \leq \frac{Ci'+D-B}{A}$. All ts that satisfy $0 \leq t \leq \frac{Ci'+D-B}{A}$ are valid thread IDs. Therefor total number of symbolic evaluation steps is equal to:

   $$\sum_{i=1}^{I} \left\lfloor \frac{Ci + D - B}{A} \right\rfloor + 1 \approx \frac{C}{A} I(I+1) + \frac{D - B + A}{A} I$$

   The right-hand side expression is expressed in terms of program constants and is proportional to cost of symbolically solving $g(i)$ for every $0 \leq i \leq I$. We compute the value of right-hand side expression in advance. If the computed cost is less than a predefined threshold, we symbolically evaluate the condition and the segment of the code under its influence. Otherwise, we follow a lower-bound upper-bound approach that was discussed in Section 3.

2. $I > T$ and $\langle At + B, \leq, g(i)\rangle$.

   Let $\alpha \in \mathbb{R}$ such that α satisfies the inequality $\frac{I}{T} \leq \alpha \frac{A}{C}$. We solve $At + B$ for each $t \in \{0, 2, ..., T-1\}$. Let $At' + B \leq Ci + D$ for $0 \leq t' < T$. As a result $\frac{At'+B-D}{C} \leq i \leq I$ and the total number of computation steps is given by:

   $$\sum_{t=0}^{T-1} I - \left\lceil \frac{At + B - D}{C} \right\rceil + 1$$

   After replacing I with $\frac{\alpha AT}{C}$ in the above sum we have:

   $$\sum_{t=0}^{T-1} \frac{\alpha AT}{C} - \left\lceil \frac{At + B - D}{C} \right\rceil + 1 \approx$$

   $$\frac{T(\alpha AT - B + D + C)}{C} - \frac{AT(T-1)}{2C} \qquad (2)$$

   It is important to note that we have iterated over thread IDs. To measure shared memory bank conflicts and thread divergence correctly, for each thread ID, values of i that have invoked its execution are saved. Later we replay these i values and compute divergence and bank conflict effects. As a result the total cost will be twice the value of expression (2).

   All terms in expression (2) except α are constants. By setting this expression equal to half of the cost threshold we can determine the maximum α. Maximum value of α determines the largest I for which the system can afford symbolic evaluation.

3. $\langle At + B, \%, g(i)\rangle$.

   We solve $At + B$ for each $t \in \{0, 2, ..., T-1\}$. Let $(At' + B) \equiv 0 \pmod{Ci + D}$ for $0 \leq t' \leq T$ which implies $Ci + D$ is some combination of prime factors of $At' + B$. If $AT + B$ is not a very large number we can store combinations of prime factors of all number less that $AT + B$ in a lookup table in order to retrieve them efficiently. Let P be a combination of prime factors of $(At' + B)$. Factor P is equal to $Ci + D$ iff $\frac{P-D}{C}$ is an integer and $0 \leq \frac{P-D}{C} \leq I$. This makes t' a solution that satisfies the condition. The total number of steps to test all values of t is bounded by:

   $$\sum_{t=0}^{T-1} \left\lfloor \frac{At + B}{2} \right\rfloor \approx \frac{AT(T-1)}{4} + \frac{BT}{2}$$

   Similar to case 2, this method requires two passes to collect performance information.

Thread ID (t')	$2t'+2$	Prime Factor Combinations (P)	Solutions $i = \log_2 \frac{P}{2}$
0	2	$\{2\}$	$\{0\}$
1	4	$\{2, 4\}$	$\{0, 1\}$
2	6	$\{2, 3, 6\}$	$\{0\}$
3	8	$\{2, 4, 8\}$	$\{0, 1, 2\}$
4	10	$\{2, 5, 10\}$	$\{0\}$
5	12	$\{2, 3, 4, 6, 12\}$	$\{0, 1\}$
6	14	$\{2, 7, 14\}$	$\{0\}$
7	16	$\{2, 4, 8, 16\}$	$\{0, 1, 2, 3\}$
.	.	.	.
.	.	.	.
.	.	.	.
127	256	$\{2, 4, 8, 16, 32, 64, 128, 256\}$	$\{0, 1, 2, 3, 4, 5, 6, 7\}$

Table 2. Symbolic Evaluation Example

The thread ID dependent condition in the partial kernel code that we discussed earlier is similar to case 3. Table 2 demonstrates the symbolic evaluation steps for some of the thread IDs of a thread block. Notice that $g(i) = 2 \times 2^i$ is a geometric function in this case. Since $2 \times 128 + 2 = 258$ is a small number, we can efficiently look up combinations of prime factors of $2t' + 2$ for each $t' \in \{0, 1, 2, ..., T-1\}$. If $2t' + 2$ has a prime factor combination P, such that $i = \log_2 \frac{P}{2}$ is an integer and $0 \leq i \leq 7$, thread t' is active during step i of the computation.

Table 2 shows all computation steps during which thread t' is active for highlighted PDG nodes in Figure 2. We sort the solutions by computation steps as shown in Table 3. Given the thread IDs that are active during each step of computation, we can determine the number of active warps and pattern of accesses to shared memory banks in each step. Values that are computed in third and fifth columns of Table 3 are used to compute execution weight of the $True$ region of the predicate node (cc) and average bank conflict penalty for shared memory accesses under the predicate node (cs).

Computation Step (i)	Thread ID (t')	Number of Warps with at Least One Active Thread	Bank Conflicts	Cycles to Resolve Shared Memory Bank Conflicts
0	$\{0, 1, 2, ..., 127\}$	4	two-way	4×8
1	$\{1, 3, 5, 7, ..., 127\}$	4	two-way	4×8
2	$\{3, 7, ..., 127\}$	4	two-way	4×8
3	.	4	two-way	4×8
4	.	4	two-way	4×8
5	.	4	no conflicts	4×4
6	.	2	no conflicts	2×4
7	$\{127\}$	1	no conflicts	1×4
$Total$		27		188
$\frac{Total}{Total\ Warps}$		$cc = \frac{27}{32} = 0.84$		$cs = \frac{188}{32} = 5.88$

Table 3. Replay of Active Threads in Each Step

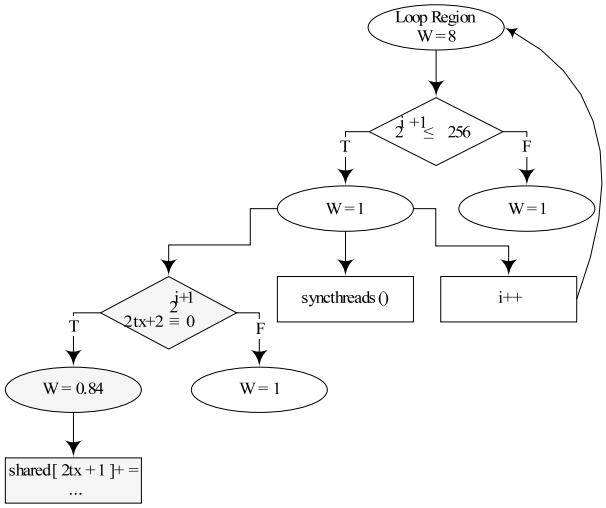


Figure 2. A Simple Program Dependence Graph

To identify the thread ID dependent effects we perform a preorder walk on the PDG. As we reach predicate nodes with a thread ID dependent condition, we symbolically evaluate the condition and memory access expressions through efficient techniques discussed in this section. During this process we augment descendent regions of predicate nodes with information that maps a step of computation (value of induction variable) to the set of thread IDs that are active during that step. This reduces the cost of symbolic evaluation for nested conditions.

3.2 Generating the String Model

To generate the string model we perform a postorder walk on the PDG of a kernel after performing symbolic evaluation on desired segments of PDG. As we visit each region node, we incorporate the associated weight with each region in calculation of the cost of descendent computational instructions.

A postorder walk of the PDG in Figure 2 derives the following string model for the partial kernel code that we discussed earlier.

$$\underbrace{\mathbf{C_6^{24} S^{cs} C_4^{16\times cc} S^{cs} C_1^{4\times cc} \bar{S}^{cs} B^4 C_2^8}}_{8\ times}$$

Coefficient cc is proportional to the average number of times that at least one thread in a warp executes a block of instructions while cs accommodates the average serialization affect of shared memory bank conflicts for each warp. Values for cc and cs are computed through symbolic execution of the PDG nodes that are shown in gray in Figure 2.

After replacing the computed values of cc (0.84) and cs (5.88) for a block size of 128, an *average warp latency* can be calculated by adding the average cost of each token based on the algorithm described in Column 3 of Table 1. Given the average warp latency, estimating the total execution time of a kernel is straight forward.

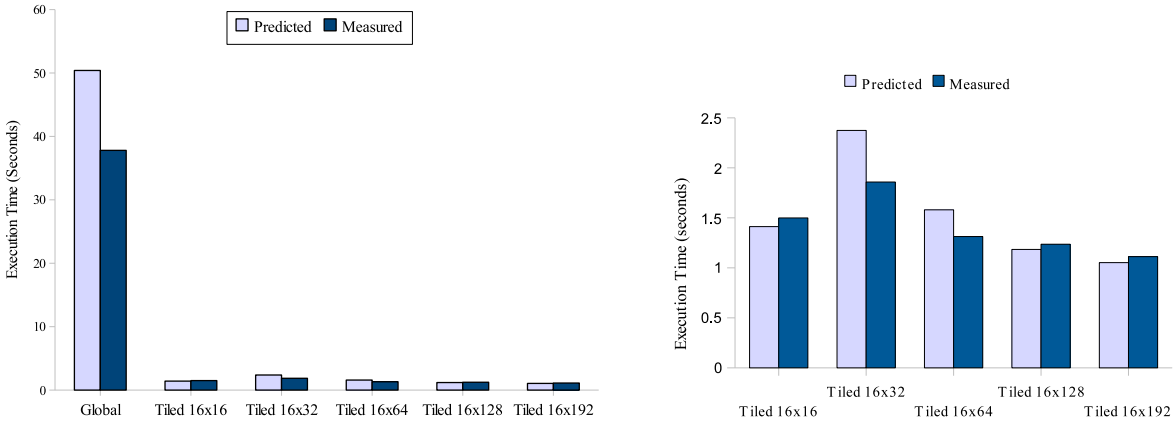
4. Evaluation

In this section we show how the performance model can be used to select the optimal configurations for three data-parallel benchmarks. We used NVIDIA GeForce 8800 GPU for our experiments.

The first benchmark, dense matrix multiplication, is representative of many tiled algorithms. We will discuss several versions of matrix multiplication and their sustained performance when multiplying two square matrices of 4K×4K elements. We begin with a simple version of matrix multiplication which exercises the global memory bandwidth of GPU. This matrix multiplication kernel creates a thread for each result element for the multiplication, for a total of 4K×4K threads. These threads loop through a sequence that loads two values from global memory, multiplies them, and accumulates the value. Figure 4(a) shows the core loops of the dot-product computation kernel; starting values for indexA and indexB are determined by block and thread coordinates. This kernel loads same array input elements multiple times by different threads. Another pitfall with this version is that global memory accesses to array A are not coalesced. In GeForce 8800, global memory delivers the 86.4 GB/s memory bandwidth only when the global memory accesses are coalesced within a half-warp. GeForce 8800 can fetch data in a single 64-byte or 128-byte transaction (NVIDIA 2007). If the memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp. As a result, the performance penalty for non-coalesced memory accesses to array A in Figure 4(a) is es-

Kernel	String Model
Matrix Multiply - Initial	$C_{11}^{44}\underbrace{\mathbf{C_2^8 M_1^{260} M_2^{260} G_1 G_2 C_4^{16}}}_{4096\ times}\ C_2^8 \bar{M}_3^{260}$
Matrix Multiply - 16×16X Tiled	$C_{12}^{48}\underbrace{\bar{M}^4 C_1^4}_{X\ times}\ C_4^{16}\ \underbrace{C_1^4}_{X\ times}\ C_4^{16}\ \underbrace{C_3^{12}}_{X\ times}\ \underbrace{c_2^8 M_1^4 G_1 \bar{S}^4}_{\mathbf{X\ times}}\ \mathbf{M_2^4 G_2 \bar{S}^4 C_2^8 B_1^4 C_1^4}\ \underbrace{C_4^{16} S^4\ \underbrace{S^4 C_2^8}_{X\ times}\ C_3^{12}\ \mathbf{B^4}}_{\mathbf{16\ times}}\ \underbrace{\bar{M}^4}_{X\ times}}_{256\ times}$
Prefix Sum Scan - Initial	$C_4^{16} M_1^4 G_1 \bar{S}^4 M_2^4 G_2 \bar{S}^4 B^4 C_6^{24}\ \underbrace{\mathbf{C_6^{24} S^{cs} C_4^{16\times cr\times cc} S^{cs} C_1^{4\times cr\times cc} \bar{S}^{cs} B^4 C_2^8}}_{\log T+1\ times}$ $C_2^8 S^{4\times\frac{32}{T}} C_2^{8\times cr'\times\frac{32}{T}} \bar{M}^{4\times\frac{32}{T}} \bar{S}^{4\times\frac{32}{T}} B^4 C_1^4\ \underbrace{\mathbf{C_6^{24} S^{cs} C_4^{16\times cr\times cc} S^{cs} C_1^{4\times cr\times cc} \bar{S}^{cs} \bar{S}^{cs} B^4 C_2^8}}_{\log T+1\ times}\ S^4 C_2^8 \bar{M}^4 S^4 \bar{M}4$
Prefix Sum Scan - Reduced Divergence and Bank Conflicts	$C_4^{16} M_1^4 C_4^{16} G_1 \bar{S}^4 M_2^4 C_4^{16} G_2 \bar{S}^4 B^4 C_4^{16}\ \underbrace{\mathbf{C_4^{16} C_4^{16\times cr\times cc} S^{4\times cc} C_4^{16\times cr\times cc} S^{4\times cc} C_1^{4\times cr\times cc} \bar{S}^{4\times cc} C_4^{16\times cr\times cc} B^4 C_2^8}}_{\log T+1\ times}$ $C_2^8 S^{4\times\frac{32}{T}} C_2^{8\times cr'\times\frac{32}{T}} \bar{M}^{4\times\frac{32}{T}} \bar{S}^{4\times\frac{32}{T}} B^4 C_4^{16}\ \underbrace{\mathbf{C_4^{16} C_4^{16\times cr\times cc} S^{4\times cc} C_4^{16\times cr\times cc} S^{4\times cc} C_1^{4\times cr\times cc} \bar{S}^{4\times cc} \bar{S}^{4\times cc} C_2^8 B^4 C_2^8}}_{\log T+1\ times}$ $S^4 C_2^8 \bar{M}^4 S^4 \bar{M}^4$

Table 4. String Models for Some of the Benchmark Kernels



(a) Initial and Tiled Kernels

(b) Zoomed for Tiled Kernels

Figure 3. Predicted versus Measured Execution Times for Matrix Multiply Kernels

timated to be 16. In other words, for accessing one word of array A GeForce 8800 issues a 16 word global memory transaction. This results in an increase in the average number of memory cycles, CYC_{mem}. Based on Equation 1, the cost of each memory operation is estimated to be of 260 cycles which is reflected in the string model representation for this kernel in Table 4.

Next, we implemented a tiled version of the the initial kernel that takes advantage of shared memory to enhance data sharing between threads computing nearby results. We chose tile sizes of 16×16, 16×32, 16×64, 16×128, 16×192 elements, to be executed by a thread block. During execution, the threads work within two input tiles that stride across 16 contiguous rows, 16, 32, 64, 128 or 192 columns in the input matrices. A total of 256 threads are created and each tied to a specific coordinate in a tile. Each thread loads the elements corresponding to its coordinates from the input tiles into shared memory. They then synchronize to establish consistency, which enables each thread to load all its inputs from shared memory. Finally, the threads calculate the partial dot product for the inputs in shared memory within a 16×X-iteration loop. Figure 4(b) shows the loops of the kernel for tile size of 16×16 while the second row in Table 4 displays the corresponding string model for 16×16X tile sizes. For the tiled kernels, global memory loads are reduced by a factor of 16, 32, 64, 128 or 192 respectively. As a result the measured performance improves as the tile size increases. Figures 3(a) and 3(b) show the predicted and

```
Ctemp = 0;

for(i = 0; i < widthA;
   i++)
{
  Ctemp += A[indexA]*
         B[indexB];
  indexA++;
  indexB +=widthB;
}

C[indexC] = Ctemp;
```

(a)

```
Ctemp = 0;
for ( ...){
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();
  for (i = 0; i < 16; i++){
       Ctemp += As[ty][i] *
          Bs[i][tx];
  }
  __syncthreads();
}
C[indexC] = Ctemp;
```

(b)

Figure 4. Partial Kernel Codes for Matrix Multiplication.(a) Initial Version (b) 16x16 Tiled Version

```
n = 2 * threadID + 1;
/*Load data into
shared memory*/
...
//use 256 threads
shared[n]+=
 shared[n-1];
__syncthreads();

//use 128 threads
if( ((n+1) % 4 == 0)
 shared[n]+=
  shared[n-2];
__syncthreads();

//use 64 threads
if( ((n+1) % 8 == 0)
 shared[n]+=
  shared[n-4];
__syncthreads();
...
```

(a)

```
#define PAD(x) ((x)+(x)/16)
/*Load data into
shared memory*/
...
//use 256 threads
shared[PAD(2*threadID+1)]+=
 shared[PAD(2*threadID)];
__syncthreads();

//use 128 threads
if( threadID < 128)
 shared[PAD(4*threadID+3)]+=
  shared[PAD(4*threadID+1)];
__syncthreads();

//use 64 threads
if( threadID < 64)
 shared[PAD(8*threadID+7)]+=
  shared[PAD(8*threadID+3)];
__syncthreads();
...
```

(b)

Figure 5. Partial Kernel Codes for Prefix Sum Scan.(a) Initial Version (b) Reduced Branch Divergence and Bank Conflicts

measured performance numbers for matrix multiply kernels next to each other. Based on these results, the proposed performance model perfectly captures global memory performance factors such as data reuse and coalescing effect.

The second benchmark we use is a power-of-two batch Fast Fourier Transform (FFT) which is based on Stockham formulation. We used parts of the pseudo-code provided in (Govindaraju et al. 2008) for implementation of FFT kernels. Figure 6(a) shows the partial kernel code for radix-R FFT that loads data from global memory, computes an R-point FFT and writes the results back to global memory. The kernel is invoked several times and during each iteration of the outer loop values from R different FFTs are combined together to generate a larger size FFT. We set T, the number of threads per block, to be $\frac{N}{R}$, where N is the input array size. For the first few invocation of this kernel writes to the global memory array A cannot be coalesced which decreases the performance.

Figure 6(b) shows another version of the kernel that we implemented to improve the data reuse by keeping intermediate results in shared memory. This version also writes the results in proper order to global memory to avoid poor global memory coalescing.

For this experiment we compute 1024 FFTs of size 256. We tried different radix sizes of 2, 4, and 16. Larger radices reduce the total number of iterations that is required to combine the the results of smaller size FFTs. Meanwhile, larger radix sizes also consume more GPU resources. For example, both global and shared memory versions for radix-16 increase use of registers substantially. Consequently array $local$ is spilled to global memory resulting in an increase in global memory traffic. Spilling array $local$ into global memory also increases the number of stall points in the kernel; and the average non-blocking cycles, NBC_{avg}, is reduced accordingly. Number of active warps, N_{warps}, is also reduced as each thread uses more registers. Reduction in both NBC_{avg} and N_{warps} makes global memory latency and pipeline delay blatant for radix-16 kernel. These effects are reflected in both predicted and measured performance numbers in Figure 7.

The last benchmark that we use is the prefix sum scan kernel, which computes partial sums of all prefixes of an array. It is related to reductions such as summation and min/max computation. The parallel scan algorithm used here is a divide-and-conquer process, due to the lack of communication support across thread blocks in CUDA. We have chosen this kernel to verify the performance model against control flow divergence, shared memory bank conflicts and synchronization overhead resulting from load imbalance. For large arrays, 4M elements in this work, the algorithm runs iteratively(Blelloch 1990), which is a common tactic in GPU programming. One kernel computes partial sums in a tiled manner, then a second kernel propagates the partial sums through the array. We will focus on the first kernel, as it occupies the majority of execution time and has a more interesting behavior from perspective of this work.

The prefix sum scan kernel illustrates the performance penalty incurred when threads within a warp take different branch paths. Unlike the other two benchmarks, computation is not distributed uniformly to threads. In mapping the algorithm to parallel threads, the tree-like structure of the computation has been divided into discrete steps containing different amounts of computation and separated by barrier synchronizations. Operations within a step are statically assigned to threads. Threads determine what to do by computing branch conditions and array indices from their thread ID. How computation is assigned to threads and the resulting branch behavior affects performance.

We start with a simple version of the scan kernel (`Init`) where thread t is responsible for the computation that produces array element $2t + 1$. Part of this kernel is shown with loops unrolled in 5(a), to illustrate the control flow and array index computation in the kernel. For a thread block size of T,

the kernel loads data from global memory, computes partial sums in $\log T + 1$ steps (3 are shown), propagates the partial sums to all array elements in another $\log T + 1$ steps, and saves the results to global memory. Subsets of the threads in a block are turned off for each step of execution, but in this organization the active threads are distributed throughout all warps for most steps of execution, leading to high value of cc in Table 4 for initial version of prefix sum scan kernel. For example, the last if-clause in Figure 5(a) restricts n to be one less than a multiple of 8, and consequently all threads that execute the following line have thread IDs that are one less than a multiple of 4.

By reassigning computation to different threads, we can group threads that take the same control flow path into the same warp, eliminating branch divergence except during the steps where fewer than one warp of threads runs. The new version (`Div`) has a lower value of cc but as the regrouped threads issue more simultaneous accesses to the same shared memory bank, cs increases from the initial version to the reduced divergence version.

We removed the shared memory bank conflicts for both of the above versions. The array layout is changed by inserting one unused element between every 16 array elements. After the array layout is changed the performance rises for the kernel with reduced branch divergence (`Div_Bank`). This kernel is shown in Figure 5(b). For the initial version performance degrades after the array layout is changed (`Init_Bank`) as the effect of reducing cs is less than the cost of adding a few extra address computation instructions in each step of computation.

Figure 8 summarizes the predicted and measured performance numbers for different versions that we discussed. For each kernel configuration we tried thread block sizes of 64, 128, 256 and 512. In general for all kernels when a thread encounters a barrier, it waits at the barrier instruction until all threads in the thread block have synchronized. After each synchronization, fewer threads are called on to do work. In fact, half the threads become superfluous after each step of computation in the first half of the scan kernel, yet they still must consume execution cycles to participate in barrier synchronization. As a result, thread blocks of larger sizes are expected to pay more penalty for synchronization in these kernel. In addition, with larger thread blocks fewer independent thread blocks are active simultaneously. As the number of active threads decreases during the last stages of tree-like computation, N_{warps} drops below the level that is required to hide the pipeline latency. The results in Figure 8 verify the accuracy of the proposed performance model to capture divergence and thread-ID-dependent effects.

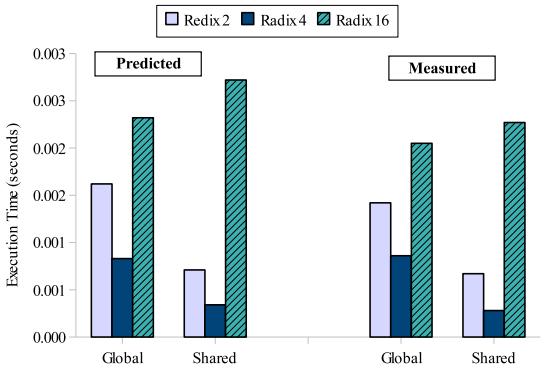


Figure 7. Predicted versus Measured Execution Times for FFT Kernels

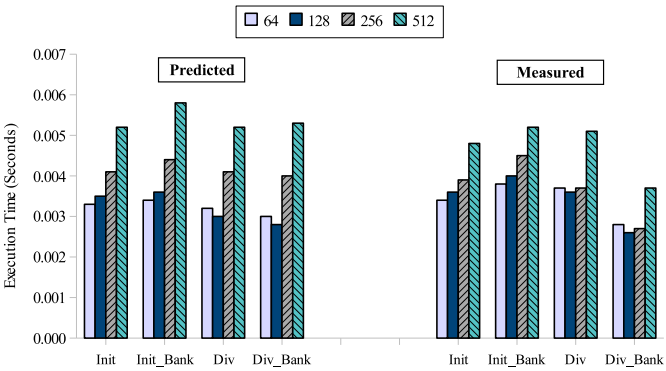


Figure 8. Predicted versus Measured Execution Times for Prefix Sum Scan Kernels

5. Conclusions and Future Work

We have presented a compiler-based approach to application performance modeling on GPU architectures. Our approach is specially attractive as it efficiently composes accurate performance information from most common C constructs. In cases that it cannot determine the precise performance for a kernel, it provides lower and upper bounds for the performance. This model allows a compiler to determine the optimality of parallel kernel configurations without running all the variations. This approach has resulted in drastically reduced kernel tuning time in our compilation process.

We validated our performance model for matrix multiply, prefix sum scan and FFT data parallel benchmarks. Our evaluation shows that there is good agreement between predicted and observed performance rankings for the various tuning versions of these kernels and the model captures the effect of all major performance factors for GPU architecture. We are in the process of fully automating the extraction of the analytical performance model from the program dependence graph of CUDA source code, which will enable us to validate the model on a much wider range of kernels. We also plan to quantify the reduction of compile time by using our analytical performance model as opposed to running the various tuning versions in our auto-tuning compilation process.

```
//Global Memory Version

for(s = 1; s < N; s*= R){
 //FFT kernel invocation here
 float2 local[R];
 n = blockID * N + threadID;
 //Load data from global memory
 for(i = 0; i < R; i++){
  indexA = n + i*T;
  local[i] = A[indexA] *
    (cos(i*W_N), sin(i*W_N));
 }
 /*Perform radix-R FFT on
   values loaded to local*/
     ...
 //Store data to global memory
 n = (n/s)*s*R + n%s;
 for(i = 0; i < R; i++){
  indexA = n + i*s;
  A[indexA] = local[i];
 }
 //FFT kernel terminates here
}
```

```
//Shared Memory Version


float2 local[R];
__shared__ work[2*N];

n = blockID * N + threadID;
//Load data from global memory
for(i = 0; i < R; i++){
 indexA = n + i*T;
 local[i] = A[indexA];
}

//radix-R FFT of size N
for(s = 1; s < N; s*= R){
 for(i = 0; i < R; i++)
  local[i] *=
   (cos(i*W_N), sin(i*W_N));
 /*Perform radix-R FFT on
   values loaded to local*/
 ...
 __syncthreads();
```

```
//Shared Memory Continued


 nd = (n/s)*s*R + n%s;
 for(i = 0; i < R; i++){
  j = nd + i*s;
  (work[j],work[j+N]) =
    local[i];
 }
 __syncthreads();

 ns = (n/T)*N + n%T;
 for(i = 0; i < R; i++){
  j = ns + i*T;
  local[i] =
   (work[j],work[j+N]);
 }
}
//Store data to global memory
for(i = 0; i < R; i++){
 indexA = n + i*T;
 A[indexA] = local[i];
}
```

Figure 6. Partial Kernel Codes for radix-R FFT.(a) Global Memory Version (b) Shared Memory Version

References

Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

MJ Clement and MJ Quinn. Analytical performance prediction on multicomputers. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 886–894, 1993.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, pages 451–490, 1991.

K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, 2004.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, pages 319–349, 1987.

Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems*, pages 85–122, 1995.

Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, 2006.

Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

Changhao Jiang and Marc Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, 2005.

T.S. Karkhanis and J.E. Smith. A first-order superscalar processor model. *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 338–349, 2004.

Weiguo Liu, Wolfgang Muller-Wittig, and Bertil Schmidt. Performance predictions for general-purpose computation on gpus. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 50, 2007.

Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13, 2004.

Marten Kongstad. *An Implementation of Global Value Numbering in the GNU Compiler Collection with Performance Measurements*, 2004.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, 2008.

NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 1.0*. NVIDIA Corporation, June 2007.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

Z. Pan and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 319–332, 2006a.

Zhelong Pan and Rudolf Eigenmann. Fast, automatic, procedure-level performance tuning. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 173–181, 2006b.

Apan Qasem, Ken Kennedy, and John Mellor-crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *In Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, pages 183–196, 2004.

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *In Proceedings of the international symposium on Code generation and optimization*, pages 204–215, 2003.

David B. Whalley. Tuning high performance kernels through empirical compilation. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 89–98, 2005.

Michael Wolfe. Beyond induction variables. *SIGPLAN Not.*, pages 162–174, 1992. ISSN 0362-1340.

Yutao Zhong, Steven G. Dropsho, and Chen Ding. Miss rate prediction across all program inputs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 79, 2003.