Supplemental Material

Loop Optimization for Divergence Reduction on GPUs with SIMT Architecture

Roman Novak, *Member, IEEE*

Appendix A
Loop Transformation Equivalence

In the following, we show the equivalence of the two loops in Fig. 1 and Fig. 3 of the main manuscript. We assume *condition1* and *condition2* being simple boolean expressions, which does not limit the applicability of the presented transformation. Note that common loop counter update or assignments in conditions, if present, can always be placed within the loop body. Further, there should be no other loop entry points and no thread synchronization requests as that would violate the initial assumption of thread independence.

The value of *condition2* needs to be known at the very beginning of the loop body in order to make the iteration delaying decision possible. Therefore, we first apply a non-aggressive loop reordering transformation. We move the first evaluation of block B1 and the initial evaluation of *condition2* out of the loop, properly guarded by the first test of *condition1*. For the 2nd and all the other iterations, we evaluate block B1 and *condition2* at the end of the loop body, also subject to *condition1*. Next, a scheduling block and the iteration-delaying divergence are inserted at the beginning of the loop body.

The presented reordering of statements does not change the relative order of any two statements, thus preserving control and data dependencies. The computations of the rearranged statements proceed through exactly the same set of states as in the original loop, which is the strongest definition of the transformation equivalence. Further, the inserted scheduling block and the iteration-delaying **if** do not change any of the original variables, while also guaranteeing loop completion through the scheduling algorithm. Therefore, the transformed loops must yield the same output given the same initial state.

• *R. Novak is with the Department of Communication Systems, Jožef Stefan Institute, SI-1000 Ljubljana, Slovenia.*
E-mail: roman.novak@ijs.si.

TABLE B.1
Up to 3-segment Optimal Fixed Schedules

p	$\bar{t}$	efficiency	schedule
0.05	1.28	0.78	ABBBBBB
0.06	1.31	0.76	ABBBBB
0.08	1.35	0.74	ABBBB
0.12	1.41	0.71	ABBB
0.18	1.47	0.68	ABBBABBBABB
0.19	1.48	0.68	ABBBABBABB
0.2	1.48	0.68	ABB
0.3	1.53	0.65	ABBABBAB
0.4	1.52	0.66	AB
0.5	1.50	0.67	AB

Appendix B
Optimal m-Segment Fixed Schedules

Let $x_1, y_1, x_2, y_2 \ldots x_m, y_m$ be an m-segment schedule dictating execution of x_i A paths followed by y_i B paths in segment i. One could find an optimal m-segment fixed schedule for a given p by an integer multivariate optimization of the thread's expected running time

$$\bar{t}_{x_1,y_1,x_2,y_2\ldots x_m,y_m} \doteq n\Big(\frac{\sum_i x_i + \sum_i y_i}{\sum_i x_i}p^2 + \frac{\sum_i x_i + \sum_i y_i}{\sum_i y_i}(1-p)^2 + \frac{\sum_i x_i(x_i+1)}{2\sum_i x_i}p(1-p) + \frac{\sum_i y_i(y_i+1)}{2\sum_i y_i}(1-p)p\Big). \quad (1)$$

The four additive terms correspond to the estimated average delays of A following A, B following B, B following A and A following B in a thread's decision sequence, weighted by the probabilities of such events. The preceding A must occur in one of the x-iterations, introducing a unit delay in $x_i - 1$ cases and $y_i + 1$ time unit in one occasion per segment, when followed by A, giving a total delay of $\sum_i x_i + \sum_i y_i$ in $\sum x_i$ considered cases. Similar reasoning applies to

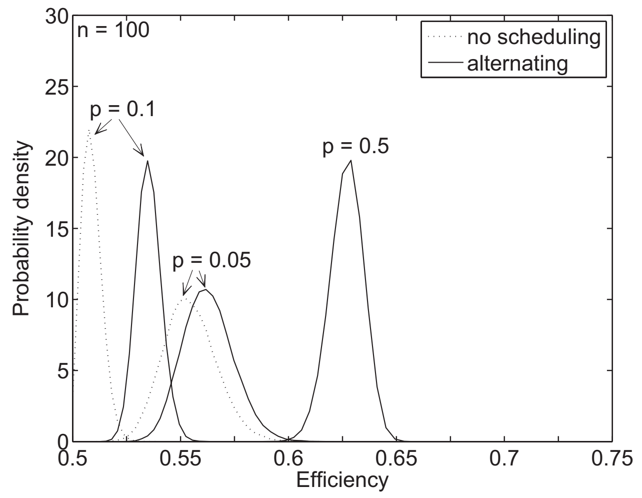

Fig. C.1. Scattering of efficiencies; representative probability density functions of a simple alternating schedule are plotted alongside the serialization reference ($n = 100$, $w = 32$).

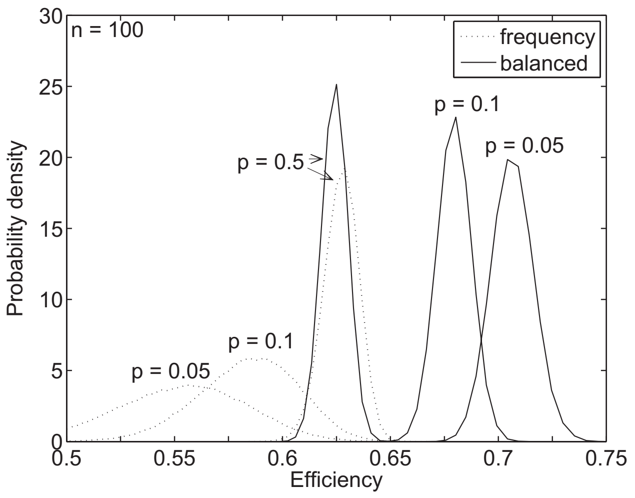

Fig. C.2. Dynamic scheduling fluctuations; lower efficiencies and severe fluctuations of the frequency scheduling at boundary p are attributed to stall conditions ($n = 100$, $w = 32$).

B following B. In B following A, and vice versa, the average delay follows upon noting that the sum of experienced delays of B's for possible occurrences of A within segment i forms a simple arithmetic series of length x_i.

Table B.1 presents up to 3-segment optimal fixed schedules for the selected branching probabilities, which were identified by finding a minimum of (1) for $x_i, y_i \in \mathbb{Z}(10)$. The expected running time is normalized to a single iteration, an inverse of which, under the assumptions given in Section 4 of the main manuscript, gives the fraction of time a SIMD unit is not idle.

Appendix C
Distribution of Efficiencies

A supplement to the efficiency analysis in Section 6 of the main manuscript presents the distribution of efficiencies in the Monte Carlo simulations. The associated probability density functions for several values of p at $n = 100$ are given in Fig. C.1 and Fig. C.2. While the running time of the native serialization is constant at non-boundary values, the iteration scheduling experiences fluctuations at all p but 0 and 1. Note that the constant time, which would show itself as a single spike at 0.5 efficiency, is not shown in Fig. C.1 for the native serialization curve. In Fig. C.2, the frequency scheduling efficiencies fluctuate severely at boundary values, whereas the balanced scheduling manages to reduce the uncertainty in that particular region at even higher efficiencies.

The uncertainties appear to combine in a quadrature, resulting in less pronounced variations at larger n, which can be observed in Fig. C.3 and Fig. C.4. Further, a shift towards higher expected efficiencies is noticeable in case of the dynamic schedules.

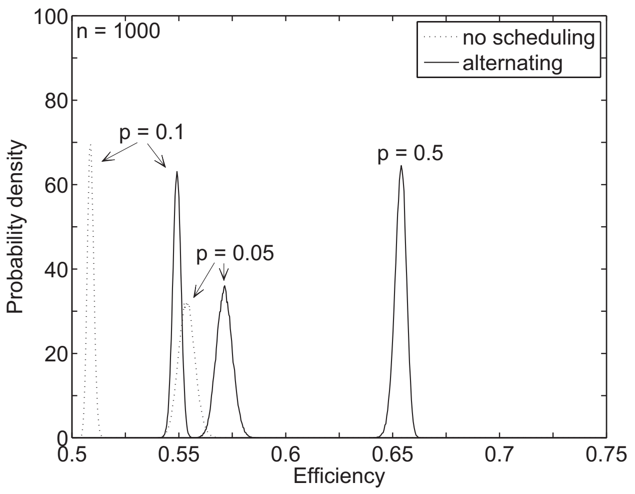

Fig. C.3. Reduced scattering of efficiencies at higher number of iterations ($n = 1000$, $w = 32$).

Appendix D
Verification Tests

The verification tests of the three scheduling techniques on a NVIDIA GPU were designed to check the derived performance limits and overhead implications on a real hardware. The paper deals with the SIMD group utilization alone, for which no profiler counter exists in the selected GPU. The real GPU environment is extremely complex and a number of factors may influence the total kernel time, hence introducing the measurement noise. Therefore, cycle counters were inserted in a loop with no global memory access and with a random sequence dictating branching decisions inline as shown in Fig. D.1. The random numbers come from the linear congruential generator. The branch processing time can be varied

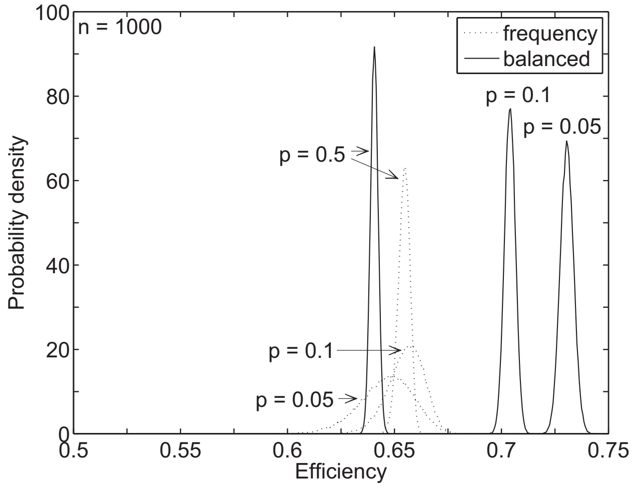

Fig. C.4. A shift towards higher expected efficiencies and a reduction in variations at higher number of iterations can be observed for the dynamic schedules ($n = 1000$, $w = 32$).

```
$rnd = seed\ [threadIdx.x]$;
**while** $i < n$ **do**
    $rnd = (0x0019660D * rnd + 0x3C6EF35F)\ \&$
            $0x00FFFFFF$;
    $path = 100 * rnd < 0x01000000 * p$;
    **if** $path$ **then**
        **for** $j = 1$ to $delay$ **do**
            $result = 3 * result + 5$;
    **else**
        **for** $j = 1$ to $delay$ **do**
            $result = 7 * result - 1$;
    $i{+}{+}$;
```

Fig. D.1. Verification kernel loop used in the SIMD efficiency measurements on a NVIDIA GPU shown without cycle counters; a single block with 32 threads should be executed to reduce the measurement noise.

and only SIMD arithmetic operations are used. The code is shown without the cycle counters and before the loop transformation. Further, p must be input as a percentage.

The GPUs used were GeForce GTX580 and GeForce GTX780, based on the Fermi [1] and Kepler [2] architectures, respectively. In the selected GPUs threads execute in a SIMD fashion. A group of 32 threads is managed as a unit during an instruction dispatch to a group of arithmetic cores, load/store units or special function units, which are functional equivalents of our SIMD processing units. Only a single block of 32 threads was executed repeatedly in order to further eliminate the measurement error. The average of the measurements was compared to the expected SIMD running time derived in Section 6 of the main manuscript, taking into account the measured t_{A}, t_{B}, t_{L} and t_{S}.

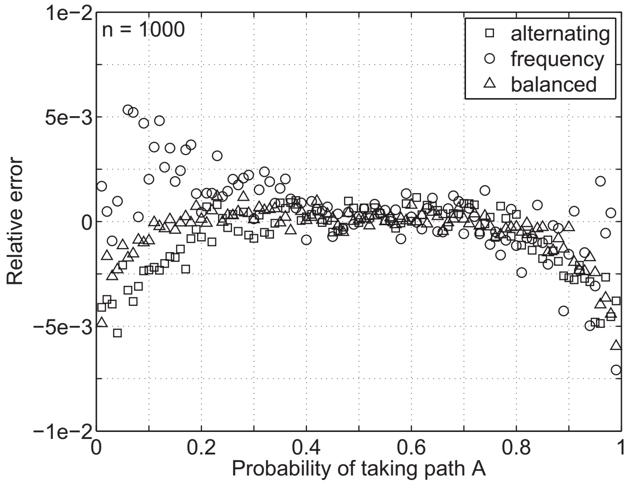

Fig. D.2. Efficiency deviations of the verification kernel relative to the expected efficiencies.

Fig. D.2 presents relative errors of the measured efficiencies with respect to the expected SIMD efficiencies on the latest Kepler architecture. The relative errors were computed for t_{A} and t_{B} being equal $2.5\,\mu$s and the overhead t_{L} of $0.12\,\mu$s. The scheduling times for the alternating, frequency and balanced scheduling used in the predictions were set to $0.05\,\mu$s, $0.08\,\mu$s and $0.09\,\mu$s, respectively. The $delay$ was set to 100 when the above values were measured. The measured efficiencies closely match the predicted values. Except from different execution times, the older Fermi architecture behaves similarly.

Appendix E
Real-World Examples

The GPU-MCML [3] is the first application to which the iteration scheduling was applied. The application assists in studying optical properties of tissues for diagnostic and therapeutic uses in medicine. A large number of photon paths are followed through the tissue layers in small steps, taking into account each layer's macroscopic optical properties while simulating phenomena such as absorption, scattering, reflectance, and transmittance. Fig. E.1 shows a simplified flowchart of the GPU-MCML kernel that reveals a loop enclosing a two sided divergence. The divergence condition checks whether a photon hit a boundary of the current tissue layer, taking either the transmit/reflect path or the absorb and scatter path. Some pre- and post-processing steps increase the loop overhead, reducing the potential gains of the iteration scheduling techniques; nevertheless, a speedup could still be observed.

A skin model provided with the source code was used in the experiments. Different hit probabilities were simulated by varying the number and the thicknesses of the layers. The original 7-layer model gave

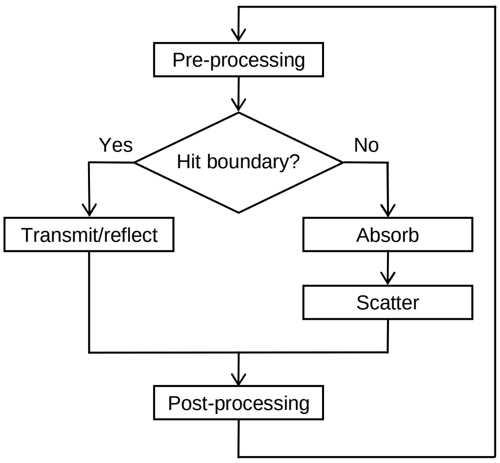

Fig. E.1. GPU-MCML flowchart emphasizing a loop with the targeted divergence.

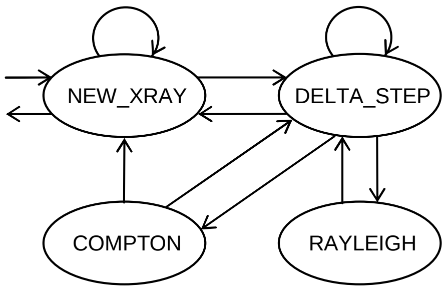

Fig. E.2. State transitions of the MC-GPU v1.1 kernel.

0.04 probability p. 0.16 and 0.44 probabilities followed after the removal of 3 and 5 layers, respectively, starting from the deepest layer, whereas 0.02 probability was achieved by increasing the thicknesses of the layers in the original 7-layer model.

Next, the iteration scheduling techniques were applied to the X-ray transport simulation code, running on a GPU and developed at the U.S. Food and Drug Administration, Center for Devices and Radiological Health [4]. The MC-GPU visualizes structures inside the human body by tracing X-rays in the voxelized geometry of a human phantom, while simulating interactions of X-ray photons with matter. The synthetic computed tomography scans can be used for evaluation of medical radiographic imaging systems or in the radiation dose estimates. The source code is in the public domain and is one of the well-known examples of the GPU efficient software [5].

The computationally intensive kernel **track_particles** from the MC-GPU v1.1 can be rewritten as a loop containing a single **switch** statement implementing the four-state machine from Fig. E.2. The state transitions are governed by the inelastic Compton scattering, elastic Rayleigh scattering, X-ray source model, phantom anatomy, X-ray detector position, and the delta scattering algorithm.

Testing was performed on the two publicly available voxelized geometries. The first, denoted as 6voxels, defined a simple 30×10×30 cm^3 object composed of six voxels, with two materials and three different densities. The second geometry was the full-body adult male phantom by Zubal et al. [6], developed at Yale University School of Medicine. The phantom contains almost 4 million voxels describing organs using 15 different organic materials. The X-ray simulations were performed at usual 90keV energy and at the lower 5keV energy. The latter configuration gives marginal probabilities of the Rayleigh and Compton steps, which allows simulations in probability regions where the balanced scheduling is expected to compensate for the frequency scheduling underperformance.

TABLE E.2
MC-GPU Branching Analysis on GTX780

	6voxels 5keV	6voxels 90keV	Zubal 5keV	Zubal 90keV	
state	p	p	p	p	t_X
NEW_XRAY	0.1910	0.18	0.00260	0.04	1 μs
DELTA_STEP	0.8073	0.59	0.99736	0.85	1 μs
RAYLEIGH	0.0014	0.02	0.00003	0.01	5 μs
COMPTON	0.0003	0.21	0.00001	0.10	10 μs

The GPU-MCML kernel was executed in 30 blocks of 512 threads, with each thread performing 50,000 loop iterations. On the other hand, the number of iterations in the MC-GPU is controlled by the number of X-ray histories. 2,500 and 1,000 histories executing in 625 blocks of 64 threads and in 1536 blocks of 64 threads were used in the 6voxels and Zubal tests, respectively. The occupancy, which is a measure of a thread level parallelism imposed by the architecture and highly sensitive to a register or shared memory usage, did not change between the tests.

In order to assess how well the time measurements agree with the projected SIMD speedup, we evaluated the branching probabilities and estimated their average processing times. Here we must emphasize that projected SIMD speedup can only be used as an indication of expected compute-bound kernel speedups. The MC-GPU branching analysis is given in Table E.2, whereas the GPU-MCML branching probabilities are given as test name suffixes. The two branches in GPU-MCML have approximately the same processing time.

The state transition dependencies in both applications are subject to some regular patterns due to inherent loss of a photon's energy in subsequent interactions with the matter. The pattern is not accounted for in our strictly random model of transitions. However, this is the expected deviation of the presented analysis.

Appendix F Additional Discussion

A natural question to ask is whether iteration scheduling can be generalized to apply to other forms of divergences in a loop body such as nested branching and repeated divergences. If we look at slightly more complicated case, for example two **if** statements in a series, we find that the structure can be transformed to a single **switch** statement with four alternatives. The **switch** branches mirrors the actual **if** branches with an addition of control logic that preserves temporal ordering of the execution. The number of iterations after such transformation is twice the original number of iterations. Another possibility is to have a four-way **switch** with combinations of possible paths through the loop body with necessary branching condition adaptations. Apart from different overhead, both transformations give the same number of **switch** alternatives. Nested divergences may be approached similarly. One can usually flatten the nested structure to a single level **switch** statement.

The problem with these transformations, which are necessary in order to apply any kind of iteration scheduling, is in the increased overhead in addition to other highly likely segments of non-divergent code in complex loop bodies. Here it is important to observe that when run natively, the overhead affects the running time exactly once per iteration. On the other hand, iteration scheduling evaluates non-divergent code once per each alternative path, as evident from the equations (3) and (4) in the main manuscript. Note that in (3) the overhead time t_{L} is consumed for all alternatives, while in (4) it affects the average execution time n-times only. This leads to high payoff threshold ratios, which can be met only in very specific applications with branch processing significantly exceeding loop control overhead.

The frequency scheduling gives better efficiencies for non-boundary branching probabilities, whereas the balanced scheduling performs well for the entire range of branching probabilities. Thus, the range of probabilities should be the main criteria in choosing one over the other. The branching probabilities can come from dynamic profiling, where the program is instrumented to record paths taken as it runs. Based on representative input one can determine how the program likely behave in general.

The more important decision than choosing between the two dynamic scheduling algorithms is the question of payoff threshold at which iteration scheduling becomes rewarding. The task requires estimation of quantities in the equations for the expected SIMD running time presented in the main manuscript. The time spent in each divergent path, as well as the time spent in non-divergent parts of a loop can be approximated reasonably well by the instruction count, at least in compute-bound kernels. On the other hand, the expected number of iterations a SIMD is concurrently processing a divergent path depends on the selected algorithm and branching probabilities. In order to provide operable data without the necessity of repeating simulations from the paper, plots of the expected number of scheduled iterations normalized by n are provided in Fig. F.1.

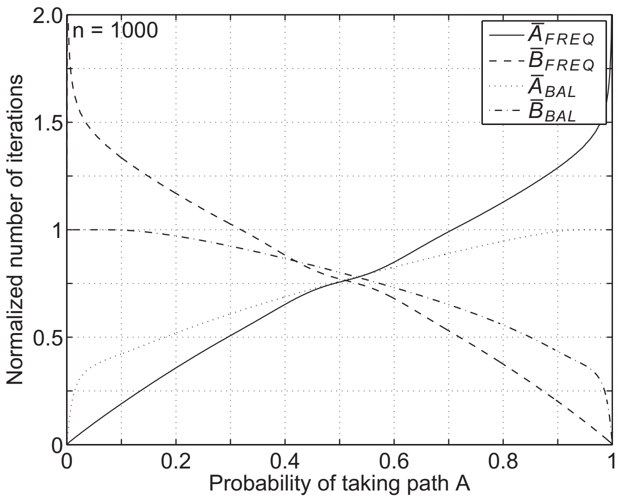

Fig. F.1. Expected number of scheduled iterations for payoff calculation; the number is normalized by n ($n = 1000$, $w = 32$).

The iteration scheduling works best at high divergent to non-divergent time ratios, which can be further increased by some loop optimization techniques [7]. Loop-invariant code should definitely be moved out of a loop [8]. Provided the scheduling functionality within a loop body remains intact, loop unrolling may reduce some overhead as well [9]. On the other hand, following optimization objectives such as maintaining locality of references [8], cache access optimization or avoiding pipeline stalls [10] may prevent decoupling of iterations within a thread group. One should consider all the advantages of a particular method, not only the SIMD efficiency. The task is far from being trivial because numerous factors should be taken into account [11]. For example, any loop transformation may affect the occupancy of NVIDIA GPUs [12]. Consequently, increasing efficiency on the thread level may result in less admitted threads for a concurrent execution on the multiprocessor level.

Studying aspects such as non-coalesced memory transactions or memory cache performance in the context of iteration scheduling on GPUs is of little or no significance. Eventual increase in the memory latency for compute-bound kernels is efficiently hidden by the fine-grained parallelism and by the availability of other active warps waiting to occupy SIMD processing units. Furthermore, extensive use of atomic operations in the rescheduled iterations may also cancel out potential gains.

References

[1] "NVIDIA's next generation CUDA compute architecture: Fermi," white paper, NVIDIA Corp., 2009.

[2] "NVIDIA's next generation CUDA compute architecture: Kepler GK110," white paper, NVIDIA Corp., 2012.

[3] E. Alerstam, W. C. Yip Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge, "Next-generation acceleration and code optimization for light transport in turbid media using GPUs," *Biomed. Opt. Exp.*, vol. 1, no. 2, pp. 658–675, Sep. 2010.

[4] A. Badal and A. Badano, "Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit," *Med. Phys.*, vol. 36, no. 11, pp. 4878–4880, Nov. 2009.

[5] W. W. Hwu, *GPU Computing Gems Emerald Edition*, 1st ed. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2011.

[6] I. G. Zubal, C. R. Harrell, E. O. Smith, Z. Rattner, G. Gindi, and P. B. Hoffer, "Computerized three-dimensional segmented human anatomy," *Med. Phys.*, vol. 21, no. 2, pp. 299–302, Feb. 1994.

[7] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2006.

[9] V. Sarkar, "Optimized unrolling of nested loops," *Int. J. Parallel Program.*, vol. 29, no. 5, pp. 545–581, Oct. 2001.

[10] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2001.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S. Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu, "Program optimization carving for GPU computing," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.

[12] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Understanding the impact of CUDA tuning techniques for Fermi," in *Proc. High Performance Computing and Simulation Int. Conf. (HPCS '11)*, 2011, pp. 631–639.