R2D2: Removing ReDunDancy Utilizing Linearity of Address Generation in GPUs

Dongho Ha
Yonsei University
Seoul, Korea
dongho.ha@yonsei.ac.kr

Yunho Oh
Korea University
Seoul, Korea
yunho_oh@korea.ac.kr

Won Woo Ro
Yonsei University
Seoul, Korea
wro@yonsei.ac.kr

ABSTRACT

A generally used GPU programming methodology is that adjacent threads access data in neighbor or specific-stride memory addresses and perform computations with the fetched data. This paper demonstrates that the memory addresses often exhibit a simple linear value pattern across GPU threads, as each thread uses built-in variables and constant values to compute the memory addresses. However, since the threads compute their context data individually, GPUs incur a heavy instruction overhead to calculate the memory addresses, even though they exhibit a simple pattern. We propose a GPU architecture called Removing ReDunDancy Utilizing Linearity of Address Generation (R2D2), reducing a large amount of the dynamic instruction count by detecting such linear patterns in the memory addresses and exploiting them for kernel computations. R2D2 detects linearities of the memory addresses with software support and pre-computes them before the threads execute the instructions. With the proposed scheme, each thread is able to compute its memory addresses with fewer dynamic instructions than conventional GPUs. In our evaluation, R2D2 achieves dynamic instruction reduction by 28%, 1.25x speedup, and energy consumption reduction by 17% over baseline GPU.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

GPU, Single Instruction Multiple Thread, redundant instruction

ACM Reference Format:
Dongho Ha, Yunho Oh, and Won Woo Ro. 2023. R2D2: Removing ReDunDancy Utilizing Linearity of Address Generation in GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589039

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589039

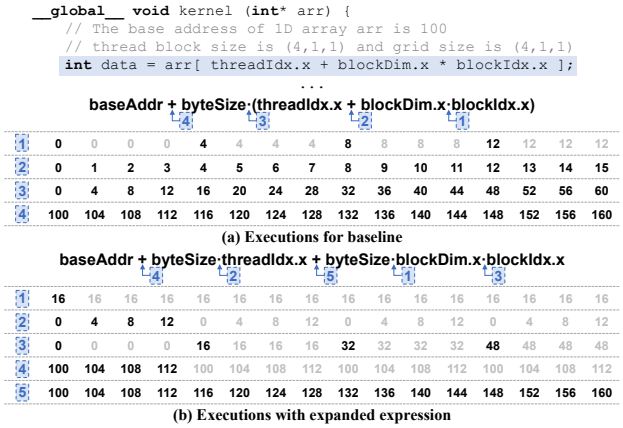


Figure 1: Examples of address calculation to show the advantages of the linearity of the SIMT. Expanding the memory address calculation clarifies and exhibits implicit redundant computation patterns.

1 INTRODUCTION

Graphics Processing Units (GPUs) provide a high throughput by employing a large number of functional units and massive multithreading [15]. To provide programmability for a massive number of threads with low overhead, GPUs employ Single-Instruction Multiple-Threads (SIMT) execution model; threads run the same static code with their own programmer-defined identifiers [29]. For instance, using NVIDIA GPUs, programmers can assign up to six-dimensional indices (three-dimensional thread block and three-dimensional grid) to threads as their identifiers. Those identifiers are available as built-in variables, such as `threadidx.x` and `blockidx.y` (we call them built-in indices). Using the built-in indices, each thread computes memory addresses to access data [1, 6, 36, 47].

In GPUs, memory address calculation and branch decisions using built-in indices introduce a large number of redundant computations across threads [16]. Figure 1-(a) shows an example of the memory address calculation, which also depicts its redundancy and inefficiency. In this example, we assume that four thread blocks (each block has four threads) run `kernel`, and all the threads access consecutive data in a one-dimensional array `arr[]`. Also, each thread computes a target memory address in the order of operator precedence. While computing `blockDim.x·blockIdx.x`, all the threads in a thread block perform identical computations with the same `blockIdx.x` value. Thus, if a GPU detects such a computation

pattern, it could remove 12 out of 64 computations (the 12 grayed numbers).

We observe that GPU kernels often calculate the memory addresses with a linear combination, a set of sums of products with built-in indices and scalar values. Figure 1-(b) shows an example of a comprehensive computation pattern detection in the address calculation; the expression is expanded from that in Figure 1-(a). No matter how many dimensions each thread block or a data structure has, hardware threads on GPUs access the data in memory whose address space is always one-dimensional. As such, a sequence of computed memory addresses across the threads often exhibits a certain degree of linearity. We call this property ***the linearity of address generation in the SIMT execution model*** (in a nutshell, ***the linearity of SIMT***). With the linearity of SIMT, we find the computation patterns of Figure 1-(b) as follows:

Scalar computations: In the first row of Figure 1-(b), `byteSize` and `blockDim.x` are constant values; all threads in the kernel use the same value when they refer to each variable. Therefore, all the threads computing the first row perform the same operation with the same source operand values. Ideally, it is enough to be done such computation by a single thread.

Computations with thread indices: In the second and fourth row of Figure 1-(b), thread index x is multiplied by `byteSize` and added by `baseAddr`. Since thread index x is repeated across thread blocks, these computations produce the same results as many as the number of thread blocks. Similar to scalar computations, GPUs need only a single thread block to compute the second and fourth rows.

Computations with thread block indices: In the third row of Figure 1-(b), thread block index x is multiplied by `blockDim.x`, which is a constant value. Since threads in a thread block have the same block index, ideally, only a single thread computation is required for a thread block.

Address calculations: In the case of the fifth row of Figure 1-(b), all the threads hold different values, unlike the above four rows. However, the values of the fifth row are the sum of the values in the fourth (calculated with thread indices) and third rows (calculated with thread block indices). Thus, if a GPU holds these values as a tuple of the fourth and third rows, GPUs can reduce the register usage; GPUs need eight values in the third and fourth rows, while 16 values are required to be kept as the results of the fifth row.

Based on the computation patterns, 29 out of 80 computations are unique in Figure 1-(b) with the expanded expression, thus requiring computing them only in the best scenario, while Figure 1-(a) computes 52 out of 64 computations. Note that, like in Figure 1-(a), the grayed elements represent redundant executions and can be removed.

Prior work also demonstrated that the SIMT execution model incurs a substantial amount of identical or affine computations across the threads using constants and built-in indices [7, 9, 18, 23, 24, 32, 34, 48, 51, 53]. To address this problem, they proposed new GPU microarchitectures or compiler techniques. However, they focused on the redundant computations without considering the linearity of SIMT, shown in Figure 1-(a), thus missing opportunities to reduce more instructions. If GPUs detected the redundant computations with the linearity of SIMT, they could reduce the total amount of computations more than prior work.

Based on the above observations, we propose a new GPU architecture called ***Removing ReDunDant Executions Utilizing Linearity of Address Generation in GPUs (R2D2)***. R2D2 eliminates the redundant executions generated by the linearity, as illustrated in Figure 1-(b), thus improving throughput and energy efficiency. The key idea is to detect the instructions performing the regular memory address calculation and pre-execute them with as many warp instructions as needed. First, the R2D2 software support detects GPU instructions producing linear combinations of built-in variables. Then, the compiler transforms such linear combinations into instructions for computing the scalar, thread-index, and block-index computations. The R2D2 microarchitecture calculates these decoupled instructions without redundancy and stores the computation results into two parts; thread-index and block-index parts. If an instruction needs the pre-computed memory addresses, R2D2 microarchitecture fetches both parts as operands. After that, R2D2 microarchitecture issues the corresponding instruction with the pre-computed operands.

In summary, this paper makes the following contributions:

- We observe that the SIMT execution model in GPUs runs a substantial amount of instructions while calculating memory addresses for all threads. Such computations tend to exhibit regular computation patterns. We call this property *linearity* of SIMT.
- We find that expressing memory addresses as linear combinations unveils implicit redundant computations and allows GPUs to remove such computations.
- We propose a GPU architecture, namely R2D2. It consists of a software technique and a new GPU microarchitecture. The software support detects the instructions that generate the linear combinations and rearranges them to explicit their redundancy. The new GPU microarchitecture executes the instructions generated by the proposed software technique.
- Our evaluation with a wide range of GPU applications shows that R2D2 can reduce the number of instructions by 28%, achieve a 1.25x geometric mean speedup, and save 17% of total energy consumption.

2 BACKGROUND AND MOTIVATION

2.1 Linearity of Address Generation

GPU kernels often start their computations by calculating memory addresses with a regular pattern. Figure 2 shows an example kernel code of the backpropagation in the Rodinia benchmark suite [6]. As shown in the code, each thread accesses four arrays (`delta`, `ly`, `w`, and `oldw`) using `index`, `index_x`, and `index_y`. Note that `ETA`, `hid`, `MOMENTUM`, and `HEIGHT` are constants. The code computes the memory addresses with built-in indices and constant variables. For example, the highlighted line in Figure 2 calculates `index` values with `(hid+1)*(HEIGHT*by+ty+1)+tx+1`. In this calculation, `by`, `tx`, and `ty` are the built-in indices representing `blockIdx.y`, `threadIdx.x`, and `threadIdx.y`, respectively. To perform the calculation, the GPU compiler translates the highlighted line to arithmetic instructions, as shown in Figure 3. As the indices and values exhibit a regular pattern, these computations often incur redundant dynamic instructions.

```
1  __global__
2  void bp_adjust_weights(float * delta, int hid, float * ly,
3                         int in, float * w, float * oldw) {
3     int by = blockIdx.y;
4     int tx = threadIdx.x;
5     int ty = threadIdx.y;
6
7     int index = ( hid + 1 ) * ( HEIGHT * by + ty + 1 ) + tx + 1;
8     //        = (hid+1)*HEIGHT*by + (hid+1)*ty + tx + (hid+2);
9
10    int index_y = HEIGHT * by + ty + 1;
11    int index_x = tx + 1;
12
13    w[index] += ((ETA * delta[index_x] * ly[index_y])
14                 + (MOMENTUM * oldw[index]));
15    oldw[index] = ((ETA * delta[index_x] * ly[index_y])
16                   + (MOMENTUM * oldw[index]));
17
18    __syncthreads();
19
20    if (ty == 0 && by ==0){
21        w[index_x] += ((ETA * delta[index_x])
22                       + (MOMENTUM * oldw[index_x]));
23        oldw[index_x] = ((ETA * delta[index_x])
24                         + (MOMENTUM * oldw[index_x]));
25    }
26 }
```

Figure 2: Example of kernel source code on backpropagation in Rodinia benchmark suite [6]. The kernel computes memory addresses using the built-in indices and constant values.

```
// PTX of Rodinia BackProp (SM_60)
// Kernel: bp_adjust_weights(P0,P1,P2,P3,P4,P5)

/* PC  */    // instructions
/*0x000*/    ld.param %rd4, [P0];
/*0x010*/    ld.param %r4, [P1];
/*0x020*/    ld.param %rd5, [P2];
/*0x030*/    mov %r1, %ctaid.y;
/*0x040*/    shl %r5, %r1, 4;
...
/*0x090*/    add %r8, %r4, %r3;
/*0x0a0*/    mad %r9, %r6, %r7, %r8;       // index
...
/*0x170*/    mul %rd13, %r9, 4;            // byte offset
/*0x180*/    add %rd14, %rd3, %rd13;       // base address
/*0x190*/    ld.global %f3, [%rd14+0];     // oldw[index]
/*0x1a0*/    cvt %fd4, %f3;
/*0x1b0*/    mul %fd5, %fd4, 0d3FD33333;
/*0x1c0*/    fma %fd6, %fd2, %fd3, %fd5;
...
```

Figure 3: Example of kernel instruction stream on backpropagation in Rodinia benchmark suite [6]. The calculated linear combinations are used as the memory addresses.

Since GPU memory has a one-dimensional address space, address calculations with multi-dimensional built-in indices require threads to compute linear combinations. For example, the highlighted line in Figure 2 can be expressed by `(hid+1)* by+(hid+1)*ty+tx+(hid+2)`. Also, likewise, `index_x` and `index_y` can be expressed as different linear combinations. We call such aspect a ***linearity of address generation in SIMT execution model*** or ***linearity of SIMT***.

The linear combinations in the memory address computations commonly consist of four types of variables; built-in indices, immediate constant, kernel parameters, and kernel sizes. For example, in Figure 3, the instruction at PC `0x030` has a built-in index `ctaid.y` that is the block index y value and the instruction at `0x040` has an immediate constant 4 as a source operand. The instructions at PC `0x000`, `0x010`, and `0x020` perform parameter load operations to get kernel input parameters. Lastly, if a kernel needs its thread block size or grid size, using special variables, such as `ntid.x` and `nctaid.y`, it loads the grid and thread block dimensions.

The linear combinations can be detected by tracking the four types of variables in the instruction stream. In Figure 3, the computation result of the instruction at PC `0x030` is by in a linear combination of built-in indices. The by value is stored in the register `%r1`. If GPUs know how `%r1` is represented with the linear combination, they can compute the result of the instruction at `0x040` as `16*by`. Thus, the output of the instruction at `0x0a0` can be represented by `16*(P1+1)*by+(P1+1)*ty+tx+P1`, which corresponds to the value of `index` in Figure 2. With this observation, we argue that it is possible to detect linear combinations by analyzing instructions and exploiting their computation patterns.

Exploiting the linearity of SIMT may result in a substantial reduction of dynamic instructions, as mentioned in Section 1. For example, in Figure 3, `P1`, `P1+1`, and `16*(P1+1)` can be done by a single thread computation. The thread-index part `tx+(P1+1)*ty` can be performed by a single block, and the block-index part `P1+16*(P1+1)*by` can be computed once per block. Finally, if an ideal GPU exploits the linearity, it can complete the same part (from `0x000` to `0x0a0`) only with 9% of computations compared to the baseline in our evaluation.

2.2 Redundancy on GPUs

Prior work demonstrated that the SIMT execution model incurs a substantial amount of identical or affine computations across GPU threads [7, 9, 18, 23, 24, 32, 34, 48, 51, 53]. To address this problem, the prior work proposed two solutions; removing redundant thread computations within a warp and redundant warp computations within a thread block.

In Figure 4, we quantify how many dynamic instruction executions are removed if a GPU ideally removes redundant thread instructions within a warp (WP), warp instructions within a block (TB) or does it by exploiting the linearity (LN). In our evaluation, ideal machines (WP, TB, and LN) execute 27%, 22%, and 33% fewer dynamic thread instructions on average compared to the baseline architecture.

Unlike the baseline GPU that executes 32 thread instructions (equivalent to a warp instruction), the WP GPU uses a single thread instruction when all the 32 threads compute the same operation with the same source operands. Prior work could reduce some redundant instructions with a scalar pipeline [18, 34]. However, they could not detect redundant computations with parameters regarding kernel and thread block sizes (or dimensions) because they are unknown before launching a kernel. Prior work represented by WP GPUs proposed various techniques to address this issue [7, 9, 23, 32, 51]. Note that the WP GPU in our experiments ideally skips all scalar computations, even if the computations require runtime information. Even if the WP GPU detects and computes the warp instructions that the scalar pipeline can cover (we call scalar warp instruction), the warp instructions should pass all GPU pipeline stages. Thus, the scalar pipeline is advantageous only if SMs should read a single thread register to compute the warp instructions, and a single lane can compute the warp instruction with less energy consumption.

Prior work demonstrated that the GPUs compute redundant warp instructions and removing such dynamic warp instructions results in throughput improvement and energy consumption reduction compared to scalar pipeline [24, 48, 53]. The TB GPU in

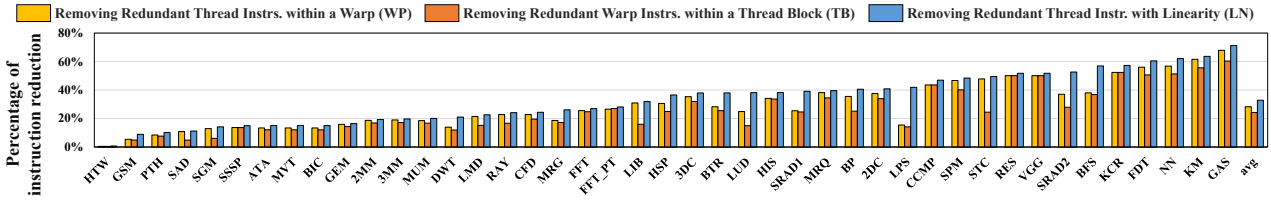


Figure 4: Dynamic thread instruction counts of the ideal machines eliminating redundant executions. WP GPUs eliminate redundant thread instructions within a warp, TB GPUs eliminate redundant warp instructions within a thread block, and LN GPUs eliminate redundant thread instructions by exploiting the linearity.

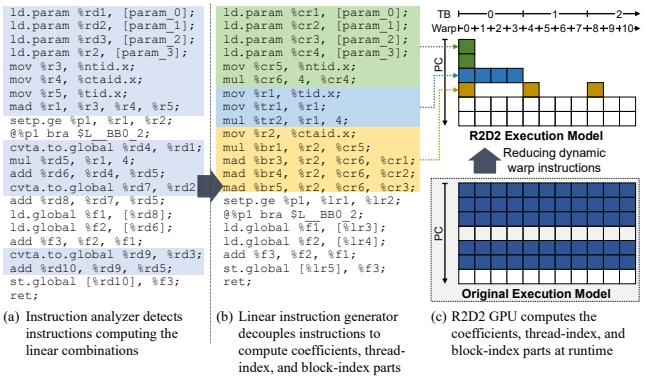
our experiments skips warp instructions that perform the same execution as prior warp instructions within a thread block. In our baseline GPUs, a thread block consists of up to 64 warps. So, ideally, the TB GPU can compute a thread block with one over 64 times fewer warp instructions.

The TB GPU should compute the warp instructions in different thread blocks even if they perform the same computation. As such, the TB GPU misses the opportunity to remove redundant warp instructions if a target workload runs tens of thousands of thread blocks. Also, the number of instructions the TB GPU removes depends on the kernel size. A small thread block size creates few opportunities in the TB GPU to remove instructions due to the few warps per thread block. Furthermore, the number of redundant computations decreases when a kernel uses one-dimensional blocks and grids compared to two or three-dimensional blocks and grids because block indices y and z mainly generate repeated operand values. Prior work targets multi-dimensional grids/blocks and proposes a TB GPUs-like approach [53].

In our evaluation, the LN GPU removes more thread and warp instructions than WP and TB GPUs. The LN GPU achieves such results due to the following three reasons. First, the redundancy addressed by WP and TB GPUs is also incurred by the linearity. Hence, exploiting the linearity of the SIMT covers the redundancy addressed by both WP and TB GPUs. Second, the LN GPU can exploit 'implicit' linearity while executing dynamic instructions. Since the compiler generates kernel instructions without considering the linearity, it forces the threads in GPUs to hold unique values in early computations, like the second row of Figure 1-(a). Thus, the redundancy of linearity does not appear in the kernel instruction stream. Third, the LN GPU can remove the redundancies within and across thread blocks. Although the thread-index part computations exhibit redundant warp instructions, these computations need to be performed only once per kernel. As GPGPU applications generally compute tens to hundreds of thousands of thread blocks, the LN GPU can take advantage of computing thread-index parts better than the TB GPU. Also, in the case of large thread block sizes, the LN GPU is advantageous over the WP and TB GPUs as the redundancy across thread blocks remains.

3 R2D2 CODE ANALYSIS AND TRANSFORMATION

To exploit the linearity of the SIMT execution model, we propose R2D2, a new GPU software support and microarchitecture design.



(a) Instruction analyzer detects instructions computing the linear combinations

(b) Linear instruction generator decouples instructions to compute coefficients, thread-index, and block-index parts

(c) R2D2 GPU computes the coefficients, thread-index, and block-index parts at runtime

Figure 5: R2D2's working mechanism. R2D2 detects and extracts the instructions computing linear combinations, and it decouples linear instructions in front of the instruction stream. Then, the SMs in the GPU computes the linear instructions without redundancy.

R2D2 detects the instructions that perform the memory address calculation and decouples them to pre-execute the memory addresses with as many warp instructions as needed. We call these decoupled instructions ***linear instructions*** and the remaining instructions ***non-linear instructions***.

Figure 5 shows the working mechanism of R2D2. First, the GPU compiler invokes R2D2's code analyzer and linear instruction generator. As shown in Figure 5-(a), the code analyzer tracks the operand registers in the program execution order and extracts the instructions generating linear combinations based on the operand dependency. After that, the linear instruction generator decouples and places detected instructions as a separate instruction stream in front of the remaining instructions, as shown in Figure 5-(b). The linear instructions consist of three instruction blocks: a block that computes scalar values for linear combinations (we call them coefficients), the thread-index part, and the block-index part. We name these three blocks (1) linear instructions for coefficients, (2) thread-index part, and (3) block-index part, respectively.

R2D2 detects redundancy and linearity in a kernel, so SMs can share the results of the computations of the linear combinations. However, SMs communication or synchronization may incur a high latency and energy overhead. Also, implementing a new interconnection structure to optimize SMs communication increases silicon area overhead and hardware complexity. To address this challenge,

we design R2D2 to make SMs compute their linear combinations individually.

Figure 5-(c) depicts how R2D2 efficiently computes the linear and non-linear instructions. The first warp (hardware warp ID `0`) executes the instructions for coefficients with the scalar pipeline. Note that the hardware warp IDs (warp entry IDs) are assigned consecutively within an SM. For example, if four thread blocks are allocated to an SM and each thread block contains 128 threads, the SM generates 16 warps whose warp IDs are from 0 to 15. Then, the first thread block (warp 0, 1, 2, and 3 in our example) executes the instructions for thread-index parts. After that, the first warps of each thread block (warp 0, 4, 8, and 12 in our example) compute the instructions for block-index parts when the thread blocks are scheduled into an SM. Such operations are done by the instruction fetch logic.

3.1 Instructions Analysis

Algorithm 1 Pseudocode of R2D2 Software

```
1:  instrs ← instructions in the kernel
2:  coefVecs ← coefficient vector table
3:  linearRegs ← linear register table
4:
5:  function R2D2_Analyzer
6:      for every instr in instrs do
7:          if instr.isListedOperation() then
8:              coefVecs ← instr.dst.coefVec()
9:              if instr.isSingleWrite then
10:                 instr.remove()
11:             else if instr.isMultiWrite then
12:                 instr.modify()
13:         for every instr.src do
14:             if instr.isLinear() then
15:                 linearRegs ← instr.src.coefVec
16:
17:     linearRegs.groupSharedIdx()
18:     for every instr in instrs do
19:         instr.modifyForBlkIdxShare()
20:
21: function R2D2_Generator
22:     for every linearReg do
23:         instrs.insertScalarInstr(linearReg)
24:         instrs.insertThreadInstr(linearReg)
25:         instrs.insertBlockInstr(linearReg)
```

This section explains how the R2D2 code analyzer inspects the code and detects the linear instructions based on Algorithm 1. The key insight of the proposed idea is that each linear combination can be represented by a coefficient vector with seven elements; one constant and six coefficients for each of the six indices because NVIDIA GPU and OpenCL programming models support up to three-dimensional thread blocks (work groups in OpenCL) and grids. We call the vectors consisting of seven elements *coefficient vectors*.

3.1.1 Lines from 6 to 10. The R2D2 code analyzer reads static instructions in program order (line #6). Then, the analyzer checks its opcode and verifies whether the operation can generate a linear combination (line #7). The operations corresponding to a part of a linear combination are `mov`, `cvt`, `add`, `mul`, `shl`, and `mad`. Figure 6 depicts how such operations generate new linear combinations with the existing ones. Once the tool detects the listed operations, it computes the destination operands' coefficient vector with the source operands' coefficient vectors (line #8). If the analyzer finds

Coefficient Vectors

src1 $\{c_1, x_1, y_1, z_1, X_1, Y_1, Z_1\}$ **src2** $\{c_2, x_2, y_2, z_2, X_2, Y_2, Z_2\}$
src2* $\{c_2, 0, 0, 0, 0, 0, 0\}$ **src3** $\{c_3, x_3, y_3, z_3, X_3, Y_3, Z_3\}$

SIMT Instruction	Conversed Operation
ld.param dst,[P]	▶ dst = {P,0,0,0,0,0,0}
cvt dst,src1	▶ dst = $\{c_1, x_1, y_1, z_1, X_1, Y_1, Z_1\}$
mov dst,src1	▶ dst = $\{c_1, x_1, y_1, z_1, X_1, Y_1, Z_1\}$
add dst,src1,src2	▶ dst = $\{c_1+c_2, x_1+x_2, y_1+y_2, z_1+z_2, X_1+X_2, Y_1+Y_2, Z_1+Z_2\}$
sub dst,src1,src2	▶ dst = $\{c_1-c_2, x_1-x_2, y_1-y_2, z_1-z_2, X_1-X_2, Y_1-Y_2, Z_1-Z_2\}$
mul dst,src1,src2*	▶ dst = $\{c_1*c_2, x_1*c_2, y_1*c_2, z_1*c_2, X_1*c_2, Y_1*c_2, Z_1*c_2\}$
shl dst,src1,src2*	▶ dst = $\{c_1*2^{c2}, x_1*2^{c2}, y_1*2^{c2}, z_1*2^{c2}, X_1*2^{c2}, Y_1*2^{c2}, Z_1*2^{c2}\}$
mad dst,src1,src2*,src3	▶ dst = $\{c_1*c_2+c_3, x_1*c_2+x_3, y_1*c_2+y_3, z_1*c_2+z_3, X_1*c_2+X_3, Y_1*c_2+Y_3, Z_1*c_2+Z_3\}$

Figure 6: List of operations that the R2D2 code analyzer targets to detect the instructions for computing linear combinations.

```
// PTX of Rodinia BackProp (SM_60)
// Kernel: bp_adjust_weights (P0,P1,P2,P3,P4,P5)

// Kernel Instructions                  Instruction Analysis
/*LINE*/ // Instructions                dst   coefficient vector
/* 01 */ ld.param %rd4, [P0];           %rd4  {P0,0,0,0,0,0,0}
/* 02 */ ld.param %r4, [P1];            %r4   {P1,0,0,0,0,0,0}
/* 03 */ ld.param %rd5, [P2];           %rd5  {P2,0,0,0,0,0,0}
/* 04 */ mov %r1, %ctaid.y;             %r1   {0,0,0,0,0,1,0}
/* 05 */ shl %r5, %r1, 4;               %r5   {0,0,0,0,0,16,0}
/* 06 */ mov %r2, %tid.y;               %r2   {0,0,1,0,0,0,0}
/* 07 */ add %r6, %r5, %r2;             %r6   {0,0,1,0,0,16,0}
/* 08 */ add %r7, %r4, 1;               %r7   {P1+1,0,0,0,0,0,0}
/* 09 */ ...
/* 10 */ mul %rd13, %r9, 4;             %rd13 {4*P1,4,4*(P1+1),0,
/* 11 */                                        0,64*(P1+1),0}
/* 12 */ add %rd14, %rd3, %rd13;        %rd14 {P5+4*P1,4,4*(P1+1),0,
/* 13 */                                        0,64*(P1+1),0}
/* 14 */ ...
/* 15 */ ld.global %f3, [%rd14+8];      %f3   -
/* 16 */ cvt %fd4, %f3;                 %fd4  -
/* 17 */ mul %fd5, %fd4, 0d3FD33333;    %fd5  -
```

Figure 7: Example of R2D2's code analysis on backprop application. The analyzer tracks the linear combinations based on built-in indices, constants, and the listed instruction in Figure 6.

that the destination operand values are linear, it removes the detected instructions from the kernel as described in line #10, as those instructions incur redundant computations and can be decoupled as linear instructions.

Figure 7 illustrates an example of how the proposed software computes the linear combinations in the Backpropagation application of Rodinia [6]. The instruction at line #5 shifts the content of register `%r1` by 4 bits to the left (equivalent to $\times 16$) and the analyzer assigns the coefficient vector `{0,0,0,0,0,16,0}` to the destination register `%r5`. Some regular values, such as kernel input parameters and dimension sizes, are unknown at compile time as they are delivered at kernel launch time. In this case, our tool writes the coefficient vectors using variable symbols. For example, the linear `%rd14` in Figure 7 can be expressed as a linear combination, and the coefficients are expressed by symbols of the second and fourth input parameters (`P1` and `P3`).

3.1.2 Lines from 11 to 12. To run a program correctly while reducing dynamic instructions, the analyzer should precisely consider control flow divergence. The NVIDIA compiler utilizes two-level instruction set architecture; an intermediate representation (PTX) and a hardware-specific instruction set (SASS). Among these two instruction set architectures, PTX instructions are written in a static single-assignment form. Each PTX instruction assigns a new register if it can, and all the registers are defined without considering

```
// Baseline Instructions
/*LINE*/ // instructions
/* 01 */ mad %r9, %r6, %r7, %r8;
/* 02 */ mul %rd4, %r9, 4;
/* 03 */ add %rd17, %rd3, %rd4;
/* 04 */ mul %rd18, %r5, 4;
/* 05 */ add %rd19, %rd17, %rd18;
/* 06 */ add %rd20, %rd19, %rd18;
/* 07 */ add %rd21, %rd20, %rd18;
/* 08 */ shl %r10, %r5, 2;
/* 09 */ add %r11, %r9, %r10;
/* 10 */ add %rd22, %rd21, %rd18;
/* 11 */ ld.global %f1, [%rd19];
```

```
// R2D2 Non-Linear Instructions
/*LINE*/ // Instructions
/* 01 */ ld.global %f1, [%lr1+%cr7];
/* 02 */ ld.global %f2, [%lr1];
/* 03 */ div %f121, %f1, %f2;
/* 04 */ ld.global %f3, [%lr1+%cr8];
/* 05 */ div %f122, %f3, %f2;
/* 06 */ ld.global %f4, [%lr1+%cr9];
```

Figure 8: Example of non-linear instruction modification to exploit the coefficient registers. The linear register values can be reused by adding coefficient values.

physical registers [10, 44]. Therefore, the divergence and loop can be detected by finding register IDs written multiple times. For example, if a kernel uses different linear combinations according to control flows, different basic blocks write different values into the same destination register ID. Also, if a basic block updates index values, the register holding the iterator is updated multiple times in the instruction stream. To exploit such a property, the R2D2 code analyzer works with the PTX instructions, and we call such registers multi-write registers.

If the analyzer meets multi-write registers in a diverged control flow, R2D2 simply computes both linear combinations and assigns them to different linear registers. Then, the code analyzer adds instructions to move the linear register data to the general-purpose register at the end of a basic block. The converted instruction allows threads to use both pre-computed linear combinations in their own control flow. Hence, R2D2 can maintain the original kernel's control flow while decoupling the instruction stream.

While executing instructions in a basic block running loops, if the analyzer detects the instructions that write new values in the already written registers, R2D2 calculates the linear registers of the first iteration of the loop and assigns a new general-purpose uniform register to hold the constant offset (line #12 in Algorithm 1). These instructions can be detected by finding instructions whose destination and one of the source registers are the same. In R2D2, the instructions update only the uniform register, which is added by the linear register values. If the offset is not a constant value, the analyzer moves linear register values to the general-purpose registers after computing the first iteration values of the loop and computes them as the original instruction stream.

3.1.3 Lines from 13 to 15. The R2D2 analyzer generates instructions that compute the linear combinations for the remaining non-linear instructions. If a destination register of an instruction is a non-linear register and another instruction uses the register as a source operand, the analyzer inserts the source operand's coefficient vector into the linear register table `linearRegs`. For example, in line #15 in Figure 7, `%rd14` that holds a linear combination is used as a source operand of the global load operation, so only the coefficient vector of `%rd14` is used to generate decoupled instructions.

3.1.4 Lines from 17 to 19. As some linear registers have the same thread-index or block-index parts, directly decoupling linear instructions with `linearRegs` may incur redundant computations and inefficient hardware utilization. For example, the thread-index part can be shared if multiple coefficient vectors have identical thread-index parts, like `w[index]` and `oldw[index]` in Figure 2. Furthermore, Figure 8 shows the middle instruction streams in the CFD application of Rodinia benchmark suite [6]. In line #5 to #7, coefficient vectors of `%rd17`, `%rd19`, `%rd20`, and `%rd21` have the same coefficient values except for the constant coefficient terms. Thus, the analyzer organizes the coefficients for linear registers to share their coefficients (line #17 in Algorithm 1). The analyzer compares linear registers' thread-index and block-index parts and groups them. To deal with this issue, the table of linear registers shapes a three-dimensional data structure; thread-index parts, block-index parts, and register IDs.

When linear registers have the same thread-index and block-index part values, R2D2 adds the difference between constant parts to one linear register value. To support such a case, the R2D2 analyzer modifies the instructions (line #19 in Algorithm 1). The modified instructions add the linear register and coefficient register values.

3.2 Instructions Decoupling

After scanning all instructions and organizing linear registers, the R2D2 instruction generator inserts linear instructions for coefficients, thread-index parts, and block-index parts based on each coefficient vector as shown in lines from #21 to #25 of Algorithm 1. The key idea is that the linear instruction generator newly defines four types of registers; thread-index register `%tr`, block-index register `%br`, coefficient register `%cr`, and linear register `%lr`. The thread-index and block-index registers hold the thread-index and block-index parts of the linear combinations, respectively, and the coefficient register holds the coefficient values. Lastly, the linear register `%lr` represents the combination of thread-index and block-index registers.

3.2.1 Line 23. Figure 9 shows how a single coefficient vector generates decoupled linear instructions in R2D2. First, the linear instruction generator writes the linear instructions to compute the coefficients of the thread-index and block-index parts, which require kernel launch time information. Like the baseline GPUs, the instruction block begins from parameter load, and the following instructions compute the coefficients in coefficient vectors. Note that the linear instruction generator does not generate instructions if the coefficient is zero or immediate values.

3.2.2 Line 24. The linear instruction generator writes the linear instructions for the thread-index part after the instructions for coefficients. The instructions fetch built-in index values to general-purpose registers and compute thread-index parts with multiply-and-add instructions. A single coefficient vector can generate up to three multiply-and-add instructions for the thread-index parts. The computation results are stored in the thread-index registers `%tr`. At line #10 in Figure 9, the instruction for thread-index parts uses coefficient register `%cr19`. Since the coefficient register is a thread register, the corresponding instruction performs a vector-scalar multiply-and-add operation. The R2D2 instruction generator writes annotation `.tr` to `mad` instruction to indicate that the second source operand is a thread register.

```
// Coefficient Vector:
// %lr3 {P5+4*P1,4,4*(P1+1),0,0,64*(P1+1),0}

/*LINE*/  // instructions
/* 01 */  // Linear instructions
/* 02 */  load.param %cr17, [P1];
/* 03 */  load.param %cr18, [P5];
/* 04 */  mad %cr19, %cr17, 4, 4;
/* 05 */  mad %cr3, %cr17, 4, %cr18;
/* 06 */  ...
/* 07 */  mov %r1, %tid.x;
/* 08 */  mov %r2, %tid.y;
/* 09 */  mad.tr %tr3, %r1, 4, 0;
/* 10 */  mad.tr %tr3, %r2, %cr19, %tr3;
/* 11 */  ...
/* 12 */  mov %r3, %ctaid.y;
/* 13 */  mov.br %br, %cr1;
/* 14 */  mad.br %br, %r3, %cr9, %br1;
/* 15 */  // Non-Linear Instructions
/* 16 */  ld.global %f1, [%lr1+4];
/* 17 */  cvt %fd1, %f1;
/* 18 */  mul %fd2, %fd1, 0d3FD33333;
/* 19 */  ...
```

Linear instructions for coefficients (computed by single thread)

Linear instructions for thread-index parts (computed by single block)

Linear instructions for block-index parts (computed by single warp per block)

Non-linear instructions (computed by all threads)

Figure 9: Example of decoupled linear instructions generated by the coefficient vector. A single coefficient vector generates the linear instructions for computing thread-index, block-index parts, and demanding coefficients.

3.2.3 Line 25. The linear instruction generator writes the linear instructions for block-index parts. Each block-index part requires a single thread computation for each thread block. To exploit the parallelism of the SIMD pipeline, R2D2 GPUs compute 16 block-index register values in parallel with a single warp. Each thread of the warp computes the block-index part values of different coefficient vectors. For example, the first thread of the warp computes the block-index part of `%lr1`, and the second thread computes the block-index part of `%lr2`.

As a result, the block-index register `%br` holds block-index parts of a thread block (8 bytes each). The `mov` and `mad` instructions at line #13 and #14 of Figure 9 use the coefficient register `%cr1` and `%cr9`. The operand collector units read the number of coefficient vectors of coefficient registers (eight in this kernel), `%cr1` to `%cr8` and `%cr9` to `%cr16`, respectively. The linear instruction generator annotates `.br` to distinguish how to access the coefficient registers from other instruction blocks.

3.3 Register Table Generation

Once the instructions for coefficients, thread index, and block index are decoupled, the PTX instruction stream is transformed into SASS instructions as the baseline. During this conversion process, the GPU compiler replaces architectural registers with linear and original instructions with physical registers. Then, the compiler counts the number of registers that each thread uses.

After writing the SASS instruction stream, the software generates a register table to combine the linear and thread-index registers. In our design, the register table has 16 entries, each consisting of eight bits; linear register ID (four bits) and thread-index register ID (four bits). Therefore, the analyzer detects up to 16 linear combinations. While decoupling linear instructions, the instruction generator combines linear registers and thread-index registers by filling in each entry's thread-index ID fields.

Also, the host writes the first PCs of instruction blocks at the kernel header. Then, with these PCs, SMs can instruct their warps to run the instruction blocks by the first thread, the first thread block, and the first warp of a thread block, respectively. We describe more details in Section 4.1.

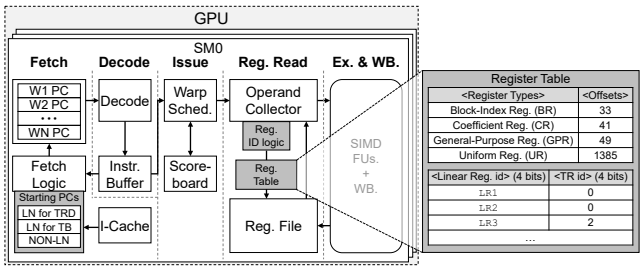


Figure 10: R2D2's microarchitecture overview. To execute the transformed kernel code, newly designed instruction fetch units fetch decoupled linear instructions. Then, R2D2 stores the linear registers in the register file and reads the linear registers referring to the register table.

4 R2D2 MICROARCHITECTURE

Figure 10 shows the R2D2's microarchitecture overview. With the new decoupled instruction stream, R2D2 computes memory addresses as follows. First, the newly designed instruction fetch units fetch decoupled linear instructions with the PCs to the 'Starting PC table'. Second, R2D2 computes the linear registers holding the pre-computed values and stores the computing results in the register file. Third, R2D2 reads the linear registers by referring to metadata kept in the register table and computing their physical register IDs. After completing the linear instructions, R2D2 executes the remaining non-linear instructions with the pre-computed linear combinations.

R2D2 stores the pre-computed linear combinations in the existing register files instead of using scalar cache structures [14] for the following reasons. First, although using cache and distributing the scalar values to SMs might save static space, additional load and store operations for the linear register values introduce a high latency overhead compared to register files and contention. Second, thread-index, block-index, and coefficient registers have different vector lengths. Thus, a cache design for those registers requires non-trivial silicon area and hardware complexity. Third, the number of linear registers varies depending on applications. Thus, if R2D2 stores them in separate SRAM, the cache structures may cause inefficient SRAM utilization.

4.1 Warp Scheduling

R2D2 GPU receives the starting PCs of linear and non-linear instructions with the kernel instruction stream. Using this information, instruction fetch logic makes the in-flight warps start their computation at different points. The first warp of an SM computes the linear instructions for coefficients, and the other warps wait until the first warp completes the coefficient computations. After the coefficient computations, the warps of the first thread block start their thread-index part computations. The instruction fetch units have a single-bit flag to indicate whether the SMs compute the coefficients and thread-index parts. Thus, the following warps do

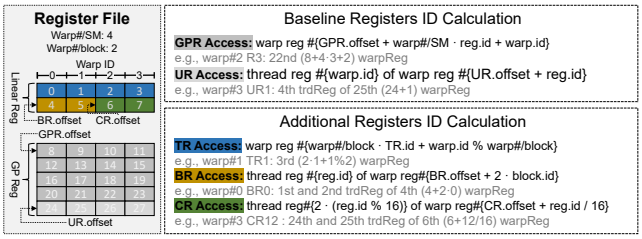


Figure 11: An example of physical register ID calculation. The operand collector units read register type offsets and register table entries to calculate the correct physical register IDs of registers.

not compute the coefficients and thread-index parts until the kernel is finished.

After that, the first warp of each thread block starts executing the linear instructions for block-index parts. The other warps need to wait until the first warp finishes the linear instructions. After running the linear instructions, the GPU can start its non-linear computations with the pre-computed memory addresses stored in the linear registers. Note that the proposed design may result in a warp synchronization issue since our baseline GPU employs the Greedy-Then-Oldest (GTO) warp scheduling policy. Thus, R2D2 issues the warps in a round-robin fashion while computing linear instructions based on the fact that prior work employed multiple warp scheduling policies at runtime [11, 31, 54]. Note that the R2D2's design presents simpler hardware compared to previous works.

4.2 Register File Access

Modern GPUs use various types of registers, such as general-purpose, predicated, and uniform registers [18]. Since GPU vendors do not provide details of the register file implementations, we optimistically assume that the types of registers are separated by type offsets indicating the first register ID of each type. Under our assumption, warps access a physical register with an ID that consists of an offset for a register type, a symbolic register ID, a warp ID, and a block size. Also, we assume that all the warps running in an SM can access the linear registers if they can compute their physical register IDs. Hence, R2D2 modifies only the register ID calculation logic while employing the same register file structure as the baseline.

In R2D2, the linear instruction generator calculates the register offsets. As mentioned in Section 3.3, R2D2 can calculate the number of linear registers needed for one thread. However, unlike the general-purpose registers, the warps in a thread block share the block-index registers, and thread blocks share the thread-index registers. Also, all threads in an SM share the coefficient registers. Hence, to compute the register type offsets, the thread block scheduler performs (the number of thread-index registers × the number of threads per block) and (the number of block-index registers per thread × the number of threads per SM) for thread- and block-index registers, respectively.

Figure 11 shows an example of how the baseline GPU and the R2D2 GPU compute physical register IDs with register offsets. The baseline GPU's operand collector units can generate source operands' physical IDs by aggregating the type offset, a product of a register ID and the number of in-flight warps, and a warp ID. First, an SM can find the physical register IDs of the thread-index registers by referring to the thread-index register field of the register table entry. The operand collector units sum up the thread-index register IDs (`TR.id`) multiplied by the number of warps per thread block (`warp#/block`) and warp ID per block (`warp.id % warp#/block`). Note that the thread-index register offset is zero.

R2D2 accesses the block-index registers by calculating the register IDs with the sum of type offset (`BR.offset`) and block IDs times two (`2·block.id`). We design such a methodology as each thread block uses two warp registers to hold block-index register values. Then, using the register IDs (`reg.id`), the physical thread register ID can be calculated. Since each linear register is coupled with block-index registers, R2D2 can access linear registers with the register IDs.

R2D2 calculates an ID of a coefficient register with the sum of type offsets (`CR.offset`) and the coefficient register IDs (`reg.id/16`). Since a single warp register holds 16 coefficients (8-byte each), computing logic divides the register id by 16. With this scheme, R2D2 uses the five least-significant bits of the coefficient register ID to find the physical thread register IDs (`2·(reg.id%16)`). The coefficients for the block-index parts and memory offsets remain in the register until the kernel finishes. R2D2 keeps the coefficients for the thread-index parts in registers until the linear instruction execution is completed. After executing the linear instructions, the R2D2 GPU uses the coefficient registers as general-purpose registers.

4.3 Memory Address Computation

The R2D2 GPU adds the thread-index and block-index parts before using them. If an instruction uses linear register values, the operations read all thread-index, block-index, and coefficient/uniform register values. Then, the R2D2 GPU stores them in the operand collector units. Existing GPUs already use instructions having more than three source operands, such as IADD3 and LOP3. Thus, R2D2 uses the same data path as the baseline GPU. After storing the thread-index, block-index, and coefficient/uniform register values, load/store units add them using an internal adder that is already equipped for immediate value addition. We believe such hardware modification incurs minor silicon overhead.

4.4 Register Usage

R2D2 should ensure that each SM has enough general-purpose registers to keep the thread-index, block-index, and coefficient registers at runtime. Prior work demonstrated that modern applications often suffer from register file shortage [21, 38, 43]. Thus, in the R2D2 GPU, if the number of thread blocks per SM is limited by the register file size limit, the host launches the original kernel instructions rather than the R2D2 instructions stream. The original instruction stream is compiled into the binary file with the R2D2's one.

R2D2 can reduce the register usage of non-linear instructions with the following scheme. First, it extracts instructions that produce linear combinations from the original instruction stream. Second, R2D2 stores memory addresses as tuples. Third, it shares thread-index parts across multiple linear registers. Moreover, GPUs launch threads to SMs in a unit of a thread block. So, the register

Table 1: Baseline GPU specification

GPU	Volta (TITAN V), 80 SMs max 64 warps/SM, max 32 block/SM
SIMD width	32
Scheduler	4 warp schedulers/SM, GTO
L1	96 KB
L2	4.5 MB, 24 way
Register File	256 KB, 8 banks per SM 14.2pJ/read 20.9pJ/write

Table 2: Benchmarks

Suite	Name	Abbr.
ispass [3]	LIB	LIB
	LPS	LPS
	RAY	RAY
parboil [47]	histo	HIS
	bri-gridding	MRG
	mri-q	MRQ
	sad	SAD
	sgemm	SGM
	spmv	SPM
	stencil	STC
polybench [42]	2DConvolution	2DC
	2mm	2MM
	3DConvolution	3DC
	3mm	3MM
	atax	ATA
	bicg	BIC
	fdtd2d	FDT
	gemm	GEM
	sesummv	GSM
	mvt	MVT
cuFFT [37]	FFT	FFT

Suite	Name	Abbr
rodinia [6]	bfs	BFS
	backprop	BP
	b+tree	BTR
	cfd	CFD
	dwt2d	DWT
	gaussian	GAS
	hotspot	HSP
	heartwall	HTW
	kmeans	KM
	lavaMD	LMD
	lud	LUD
	mummergpu	MUM
	nn	NN
	pathfinder	PTH
	srad_v1	SRAD1
	srad_v2	SRAD2
graphBig [35]	Connected component	CCMP
	K-core decomposition	KCR
	Shortest path	SSSP
Nebula [22]	ResNet	RES
	VGGNet	VGG

file space is available if it is smaller than the number of registers of a thread block. We will show a detailed evaluation of register usage in Section 5.6.

Another issue is how to reside the linear registers until the kernel finishes. Kernels often compute more thread blocks than GPUs can execute in parallel. Thread block schedulers allocate as many thread blocks as the number of the predetermined limit of thread blocks in all SMs. When an in-flight thread block finishes, the scheduler assigns the following thread blocks to the SM that ran the finished thread block. In our register file model, the following thread blocks simply use the previous thread block's warp registers without the release process because the register usages of each thread block are identical. As the linear registers are read-only and separated by general-purpose register offset, multiple thread blocks can use linear registers without additional work during the kernel execution.

5 EVALUATION

We use GPGPU-Sim v4.0 [20], GPUWattch [27], and CACTI 7.0 [50] to design R2D2 and evaluate the performance and energy consumption. We model the baseline GPU based on NVIDIA TITAN V Volta GPU. The detailed configurations are described in Table 1. The baseline GPU includes a scalar pipeline for the operations with constant variables as in existing GPUs [18, 34]. We use various benchmarks listed in Table 2.

The R2D2 software performs the compile-time instruction analysis and the linear instruction generation with the static instructions of GPGPU-Sim. Such an implementation is inspired by the prior work [48, 53]. First, the software extracts the PTX and SASS instructions from the binary file using the NVIDIA software tool `cuobjdump`. Then, the R2D2 analyzer investigates and decouples the PTX and SASS instruction stream before the actual simulation. Using the transformed kernel and metadata, GPGPU-Sim calculates statistics of the applications.

We compare R2D2 to the following GPU architectures proposed by prior work [48, 53].

Decoupled Affine Computation (DAC): DAC decouples affine and non-affine computation with compiler support and executes them [48]. We model an optimistically working DAC by computing all warp instructions producing consecutive affine values with a single warp instruction without any overhead.

Dimensionality-Aware Redundant SIMT Instruction Elimination (DARSIE): DARSIE detects redundant warp instructions and skips these computations [53]. Also, DARSIE can skip redundant load instructions if the load instructions do not occur any memory dependency problems. We model DARSIE by skipping redundant warp instructions within a thread block with no overhead. We also compare our design with DARSIE+Scalar, which can skip redundant warp instructions. If non-redundant warp instruction computes with scalar operands, the scalar pipeline handles it.

5.1 Instruction Count Comparison

Figure 12 shows the number of dynamic instructions run by DAC, DARSIE, DARSIE+Scalar, and R2D2. We normalized all the instruction counts to the baseline. Note that the applications are sorted by the percentage of instructions reduction of LN machine (Figure 4) in ascending order. R2D2 reduces the dynamic instruction count by 28% on average compared to the baseline. DAC, DARSIE, and DARSIE+Scalar reduce the instruction count by 20%, 18%, and 19%, respectively.

In most applications, R2D2 eliminates more dynamic instructions than DAC, DARSIE, and DARSIE+Scalar as it proactively utilizes the linearity of the SIMT. First, R2D2 uncovers implicit linearity by decoupling and analyzing regular computations, which both DAC and DARSIE GPUs cannot successfully detect. DAC only focuses on affinity and handles it by implementing affine computing units. DARSIE mitigates redundancy across warp instructions and skips them with compiler support and a hardware-level instruction skipping mechanism. The proposed R2D2 addresses both computation patterns covered by DAC and DARSIE via thread- and block-index registers, which achieves performance improvement in a broader range of applications.

Additionally, while R2D2 detects redundancy across thread blocks, DAC, DARSIE, and DARSIE+Scalar detect redundancy only within a thread block. As such, R2D2 can reduce more dynamic instruction count than DARSIE and DARSIE+Scalar in 2DC, STC, and SRAD2 applications. Those applications use a thread block consisting of up to 16 warps but thousands of blocks. For example, SRAD2 runs 65,536 thread blocks, and each thread block contains eight warps. In this case, since DARSIE detects redundant warps within a thread block, DARSIE can remove up to 15 dynamic warp instructions

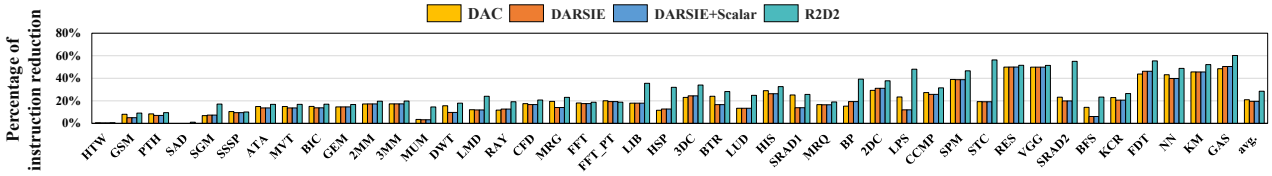


Figure 12: Percentage of dynamic warp instruction reduction compared to the baseline GPU. R2D2 reduces dynamic instruction count by 28% on average, while DAC, DARSIE, and DARSIE+Scalar reduce 20%, 18%, and 18%, respectively.

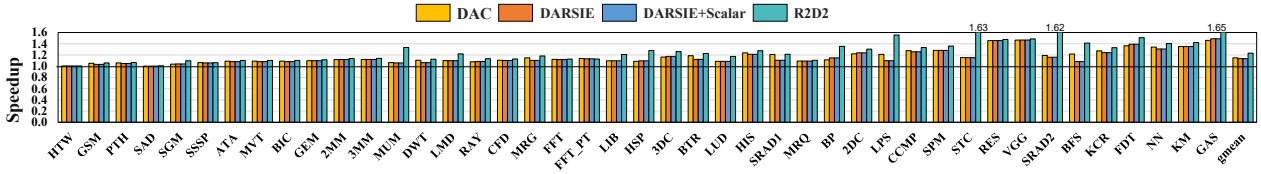


Figure 13: Speedup over the baseline GPU. R2D2 achieves a geometric mean speedup of 1.25x, while DAC, DARSIE, and DARSIE+Scalar exhibit 1.15x, 1.14x, and 1.14x speedup, respectively.

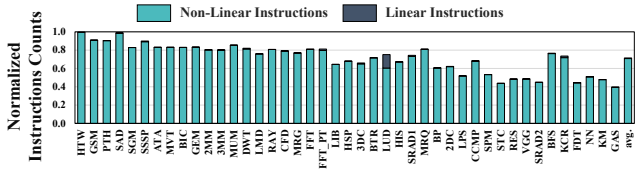


Figure 14: Dynamic instruction count of linear and non-linear instructions of R2D2 normalized by the baseline GPUs.

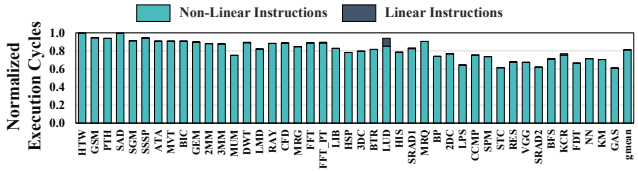


Figure 15: Execution cycles of linear and non-linear instructions normalized by the baseline GPUs.

per static instruction. Unlike DARSIE, in R2D2, 820 (65,536/80) thread blocks can share the thread-index parts. Even in applications with one-dimensional thread blocks, such as KM and FDT, R2D2 reduces the dynamic instruction count more than the previous techniques, as the applications have thousands of thread blocks.

R2D2 analyzes the static code at compile time, so predicting such runtime behavior as loops run by `for` or `while` is difficult. However, the matrix-multiply application (SGM) keeps adding constant values and linear combinations to move the computation windows, which can be covered by coefficient register usage (Section 3.1). Hence, R2D2 shows an advantage in removing redundant computations in the SGM. Additionally, since R2D2 can remove redundant computations across the thread blocks, R2D2 achieves a higher instruction reduction than prior work.

5.2 Performance

Since each approach executes different numbers of instructions to complete the applications, it is not possible to use the Instruction Per Cycle (IPC) as a throughput metric. Thus, we use the end-to-end execution time of all the architectures as the key performance metric rather than Instruction Per Cycle (IPC). Figure 13 shows the experimental results of DAC, DARSIE, DARSIE+Scalar, and R2D2 over the baseline GPU. R2D2 achieves 1.25x geometric mean speedup, while DAC and DARSIE achieve 1.15x and 1.14x. All the architectures could reduce the execution time by reducing dynamic instruction counts, shown in Figure 12.

The applications used in the evaluation exhibit various effects of the dynamic instruction count reduction on the execution time. For example, LPS and SPM show a similar degree of dynamic instruction reduction ratio, 48%, and 47%, respectively. However, R2D2 achieves different speedup results: 1.56x (LPS) and 1.36x (SPM). It is because those applications have different memory access behaviors. SPM frequently loads and stores computation results from/in the shared and device memory. As SPM is a memory-intensive workload, R2D2 achieves a smaller speedup than LPS.

R2D2 also benefits the BFS application that exhibits irregular memory access patterns. In BFS, we observe that it still contains a considerable number of load/store operations having regular memory access patterns. While running the kernels of the BFS application, a GPU performs eight memory accesses per loop, and four accesses exhibit irregularity among those eight memory operations. As such, R2D2 achieves a 1.4x speedup in this application. We also evaluate graph analysis applications (CCMP, KCR, and SSSP). For CCMP and KCR, R2D2 achieves performance improvement for the same reason as BFS; the applications often conduct regular memory accesses as many as irregular ones. However, the performance improvement for SSSP is not significant as it is more irregular than the other applications. In this case, R2D2 rarely detects redundant

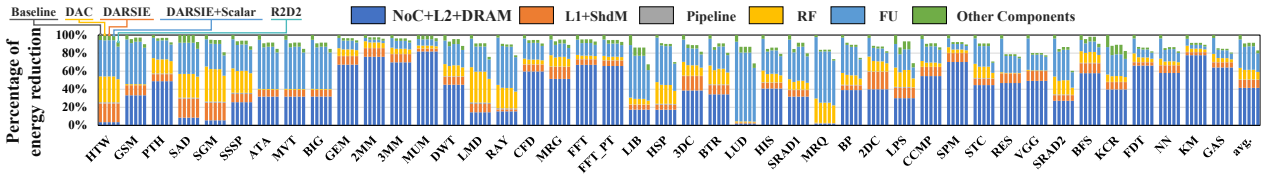


Figure 16: Energy reduction normalized to the baseline GPU. R2D2 saves 17% of energy consumption on average, while DAC, DARSIE, and DARSIE+Scalar reduce 9%, 8%, and 9%, respectively.

computations and linear combinations in their kernel, so only a few instructions are generated as the overhead of R2D2.

5.3 Overhead Analysis

Figure 14 shows the breakdown of dynamic instructions run by R2D2. All the results are normalized to the instruction counts of the baseline GPUs. In R2D2, the decoupled linear instructions for coefficients, thread-index, and block-index parts account for only 1% of total dynamic instructions on average. R2D2 shows the highest instruction overhead, 19% in LUD. In this case, the application launches tens of kernels that consist of one to hundreds of thread blocks. Therefore, such a small size of kernels decreases the opportunity to remove the redundant computations in R2D2 GPUs. However, R2D2 still shows a 25% of dynamic instruction reduction, while DARSIE+Scalar shows a 13% of dynamic instruction reduction in LUD.

Figure 15 shows the execution cycle breakdown of R2D2. The execution time of the linear instructions in R2D2 accounts for only 1% of the total execution time. Non-linear instructions should wait until all the SMs complete the linear instruction computations. As mentioned in Section 5.1, 3DC and LUD show the heaviest cycle overhead among the applications.

Moreover, note that DAC requires similar compiler support to decoupled affine and non-affine instructions but more hardware modifications to compute affine instructions efficiently. Also, DARSIE needs similar compiler support and hardware modifications to detect and remove redundant warp instructions.

5.4 Pipeline Latency Study

The R2D2 GPU incurs latency overheads in instruction fetch logic for linear instructions, physical register ID computations for linear register accesses, and additions between thread-index and block-index registers for generating memory addresses. However, R2D2 tolerates such latencies as it efficiently computes the linear instructions. So, the pipeline latencies can be hidden by thread-level parallelism.

Since GPU vendors do not publish the details of the fetch units' and operand collector units' circuit designs, we evaluate a case in which the average speedup drop reaches 1% by gradually increasing the latency overhead to demonstrate the tolerance. First, R2D2's fetch units access the starting pc table, and one-bit flag registers to fetch the linear instructions. In our evaluation, 7-cycle latency results in a 1% speedup drop. Second, the R2D2 GPU conducts more complex arithmetic operations than general-purpose registers to compute the physical register IDs of linear registers. In our evaluation, the average speedup decreases by 1% with 5-cycle latency. Third, to compute linear combinations as memory addresses, the R2D2 GPU should add a thread-index register and a block-index register value. To simulate this, we assume that the addition operation takes four cycles, the same as the add operation with the baseline CUDA cores.

5.5 Energy Consumption

Figure 16 shows the energy consumption breakdown of DAC, DARSIE, DARSIE+Scalar, and R2D2. Compared to the baseline, R2D2 reduces the energy consumption by 17% while DAC, DARSIE, and DARSIE+Scalar reduce 9%, 8%, and 9%, respectively. R2D2 reduces the operations of the arithmetic units and the register access by removing redundant dynamic instructions. Such an advantage results in the reduction of power consumption. Also, R2D2 reduces the execution time, so it achieves energy saving, as shown in the experimental results. The dynamic instruction reduction of all four techniques did not result in significant energy savings in memory-intensive workloads as memory operations consume more energy than arithmetic operations [17].

5.6 Register Usage Study

As mentioned in Section 4.2, R2D2 can run the register-bounded applications by reducing the dynamic instruction count as linear registers occupy a relatively small number of physical registers. For instance, one of the kernels `block2D_hybrid _coarsen_x` in the STC application meets register restriction while using 32 registers per thread. The kernel uses 128 threads per block, so the 256KB register file (65536 4-byte registers) of an SM can hold registers of 16 thread blocks (`65536/(128·32)`). In this case, there is no register remaining. However, since R2D2 replaces three 8-byte general-purpose registers with linear registers, 12,288 (multiplying three registers, two 4-byte, 128 thread/block, and 16 block/SM) 4-byte thread registers are available. The kernel generates four thread-index registers, 16 block-index registers, and 67 coefficient registers, so 1,091 4-byte thread registers are required for thread-index registers; 512 (4·128), 512 (`2·16·16`), and 67 for thread-index registers, block-index, and coefficient registers, respectively.

To demonstrate that R2D2 in modern register-bounded kernels is available, we evaluate graph analysis, FFT, and neural network benchmarks; CCMP, FFT, KCR, RES, SSSP, and VGG. Like the other applications, such benchmarks do not exceed the register file size restriction.

As we mentioned in Section 4.4, the R2D2 GPU computes original kernel instructions if it has no space for linear registers. In this case, two types of overhead occur; binary file size overhead to hold the original kernel code and cycle overhead to calculate

Table 3: Sensitive Study: Blocks per Grid

backprop	BP_04	BP_08	BP_16	BP_32	BP_16
Instr. Reduction	38.3	38.8	39.2	39.6	39.7
Speedup	1.35	1.35	1.36	1.36	1.36

the number of in-flight thread blocks considering linear registers. However, in GPGPU computing, it is common for binary files to hold multiple version kernels and use the just-in-time compiler to support multiple generations of hardware. We believe that the overhead of such computations is negligible.

5.7 Case Study: Persistent Thread Implementation

The persistent thread approach helps threads share some redundant inter-block computations by defining virtual threads and scheduling them via a software work queue [13]. The persistent thread implementation issues as many threads as the SMs can compute concurrently and schedules thread workloads based on their communication pattern via a work queue. Thus, the R2D2's performance depends on the communication pattern. To evaluate such an effect, we implemented two types of the Fast-Fourier Transform (FFT) benchmark; non-persistent (FFP) and persistent thread (FFT_PT) styles. If the communication pattern is regular, the application computes thread indices and conducts linear operations on them, which can be covered by R2D2. As FFT_PT uses regular communication patterns in our implementation, R2D2 shows considerable performance improvement in FFT_PT as shown in Figure 13.

5.8 Sensitivity Study

5.8.1 Blocks per Grid. We evaluate the instruction reduction and speedup in the backprop benchmark with various numbers of blocks per kernel. Varying the number of blocks is the most straightforward way to shrink or enlarge the kernel. The backprop benchmark requires an input argument indicating the number of input nodes, which can be computed in parallel. For example, `BP_04` means performing backprop with 2^4 input nodes. Since the backdrop application uses the fixed thread block size (256), the code decides the grid size with the input node number and computes the kernel. Table 3 shows the experimental results. R2D2 shows consistent performance improvement across the various number of thread blocks. R2D2 achieves such results as the number of linear instruction computations is relatively small compared to non-linear instruction computations.

5.8.2 The number of SMs. We evaluate how the number of streaming multiprocessors (SMs) in R2D2 affects its performance. We tested R2D2 with 80, 100, 120, 140, and 160 SMs while keeping the kernel size of the applications the same. As the number of SMs increases, each SM computes fewer thread blocks, so the linear combinations are shared across fewer thread blocks. However, even with 160 SMs, there is no performance drop compared to 80 SMs because the SMs can execute their linear combinations simultaneously, and the number of linear instruction computations is relatively small compared to non-linear instruction computations.

6 RELATED WORK

Instruction Skipping on CPUs: First, prior work proposed the instruction reusing or skipping techniques for CPUs [4, 5, 19, 28, 30, 33, 39–41, 45, 46, 49]. They demonstrated temporal redundancy in CPU instructions and proposed techniques exploiting the redundancy, thus improving throughput as well as energy efficiency. However, since GPUs run thousands of threads concurrently, it is challenging to efficiently reuse/skip redundant executions on GPUs.

Scalar pipeline: Prior work focused on the redundancy caused by the value locality within a warp that we have explained in Section 2 [2, 7–9, 12, 18, 23, 32, 34, 51, 52]. They demonstrated that scalar and affine warps are common in GPGPU applications. Also, they proposed a technique for detecting uniform and affine warp to reduce data transfers between the register file and functional units. Nowadays, several modern GPU architectures support the scalar pipeline [18, 34].

Warp redundancy: There are researches to eliminate the redundant warp instructions [24, 48, 53] within a thread block. Wang et al. showed that the affinity demonstrated by Collange et al. [9] can be extended to consecutive warps so that multiple warp values can be represented by the affine tuple (the first value and difference) as memory address or predicate operands [48]. Therefore, they proposed to decouple kernel code into two in compile-time; affine and non-affine instruction streams.

Yeh et al. showed that analyzing thread hierarchy and its usage in the kernel source code allows the compiler to detect redundant executions, especially when the applications use multi-dimensional thread block [53]. In DARSIE, the compiler detects redundancy based on thread x indices at kernel launch time and skips detected redundant warp instructions.

Kim et al. introduced a GPU microarchitecture detecting redundant executions and skipping them by hardware memoization [24].

Value similarity: Several works focused on another type of value similarity on GPUs. They observed that threads in a warp produce a vector with a small value range [25, 26]. To exploit such an insight, they proposed a register value compressing technique to save computation or register file energy consumption.

7 CONCLUSION

In this paper, we demonstrate that GPUs employing the SIMT execution model often calculate the memory addresses represented as linear combinations with built-in indices. Furthermore, by exploiting such linearity of address generation, it is possible to reduce a large number of dynamic instructions efficiently. We propose R2D2 to exploit the linearity of the SIMT execution model in GPU architecture and improve the throughput and energy efficiency. In the proposed R2D2, the static instruction analyzer extracts the linear combinations of built-in indices. Then, the linear instructions inserted before the original instruction stream compute the extracted linear combinations. Such computation results are stored and used via the SMs' register file. As a result, R2D2 achieves dynamic instruction reduction by 28%, a 1.25x geometric mean speedup, and total energy consumption reduction by 17%.

ACKNOWLEDGMENTS

This work was partly supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021- 0-00853, Developing Software Platform for Programming of PIM), the Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education (NRF-2022R1C1C1011021), and the Super Computer Development Leading Program of the National Research Foundation of Korea (NRF) grant funded by the Korean government (Ministry of Science and ICT (MSIT)) (NRF-2021M3H6A1017683). Won Woo Ro is the corresponding author.

REFERENCES

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. 2018. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture* 13, 2 (2018), 1–140.

[2] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. 2013. Convergence and Scalarization for Data-Parallel Architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494995

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[4] Saisanthosh Balakrishnan and Gurindar S. Sohi. 2003. Exploiting Value Locality in Physical Register Files. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, USA, 265.

[5] J. Adam Butts and Guri Sohi. 2002. Dynamic Dead-Instruction Detection and Elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/605397.605419

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[7] Zhongliang Chen and David Kaeli. 2016. Balancing Scalar and Vector Execution on GPU Architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 973–982. https://doi.org/10.1109/IPDPS.2016.74

[8] Zhongliang Chen, David Kaeli, and Norman Rubin. 2013. Characterizing scalar opportunities in GPGPU applications. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 225–234. https://doi.org/10.1109/ISPASS.2013.6557173

[9] Sylvain Collange, David Defour, and Yao Zhang. 2009. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. In *Proceedings of the 2009 International Conference on Parallel Processing* (Delft, The Netherlands) *(Euro-Par'09)*. Springer-Verlag, Berlin, Heidelberg, 46–55.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[11] Ahmed ElTantawy and Tor M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[12] Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. 2012. Power-Efficient Computing for Compute-Intensive GPGPU Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 445–446. https://doi.org/10.1145/2370816.2370888

[13] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14. https://doi.org/10.1109/InPar.2012.6339596

[14] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 608–619. https://doi.org/10.1109/HPCA.2018.00058

[15] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. 2009. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *IEEE Computer Architecture Letters* 8, 1 (2009), 25–28. https://doi.org/10.1109/L-CA.2009.4

[16] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

[17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 243–254. https://doi.org/10.1109/ISCA.2016.30

[18] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[19] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA) *(MICRO 31)*. IEEE Computer Society Press, Washington, DC, USA, 216–225.

[20] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[21] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. RegMutex: Inter-Warp GPU Register Time-Sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 816–828. https://doi.org/10.1109/ISCA.2018.00073

[22] Bogil Kim, Sungjae Lee, Chanho Park, Hyeonjin Kim, and William J. Song. 2021. The Nebula Benchmark Suite: Implications of Lightweight Neural Networks. *IEEE Trans. Comput.* 70, 11 (2021), 1887–1900. https://doi.org/10.1109/TC.2020.3029327

[23] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 130–141. https://doi.org/10.1145/2485922.2485934

[24] Keunsoo Kim and Won Woo Ro. 2018. WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 389–402. https://doi.org/10.1109/HPCA.2018.00041

[25] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Murali Annavaram, and Won Woo Ro. 2017. Improving Energy Efficiency of GPUs through Data Compression and Compressed Execution. *IEEE Trans. Comput.* 66, 5 (2017), 834–847. https://doi.org/10.1109/TC.2016.2619348

[26] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-Compression: Enabling Power Efficient GPUs through Register Compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/2749469.2750417

[27] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[28] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada) *(ISCA '00)*. Association for Computing Machinery, New York, NY, USA, 182–191. https://doi.org/10.1145/339647.339678

[29] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55. https://doi.org/10.1109/MM.2008.31

[30] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 138–147. https://doi.org/10.1145/237090.237173

[31] Jiwei Liu, Jun Yang, and Rami Melhem. 2015. SAWS: Synchronization Aware GPGPU Warp Scheduling for Multiple Independent Warp Schedulers. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/2830772.2830822

[32] Zhenhong Liu, Syed Gilani, Murali Annavaram, and Nam Sung Kim. 2017. G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 601–612. https://doi.org/10.1109/HPCA.2017.51

[33] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong. 2010. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 337–348. https://doi.org/10.1109/MICRO.2010.41

[34] Mike Mantor. 2012. AMD Radeon™ HD 7970 with graphics core next (GCN) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 1–35.

[35] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 69, 12 pages. https://doi.org/10.1145/2807591.2807626

[36] NVIDIA. 2023. Cuda samples. Retrieved Mar 30, 2023 from https://github.com/NVIDIA/cuda-samples

[37] NVIDIA. 2023. cuFFT. Retrieved Mar 30, 2023 from https://docs.nvidia.com/cuda/cufft/

[38] Yunho Oh, Myung Kuk Yoon, William J. Song, and Won Woo Ro. 2018. FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 364–376. https://doi.org/10.1109/MICRO.2018.00037

[39] Arthur Perais and André Seznec. 2016. Cost effective physical register sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 694–706. https://doi.org/10.1109/HPCA.2016.7446105

[40] V. Petric, A. Bracy, and A. Roth. 2002. Three extensions to register integration. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* 37–47. https://doi.org/10.1109/MICRO.2002.1176237

[41] V. Petric, T. Sha, and A. Roth. 2005. RENO: a rename-based instruction optimizer. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 98–109. https://doi.org/10.1109/ISCA.2005.43

[42] Louis-Noël Pouchet. 2015. PolyBench/C: the Polyhedral Benchmark suite. Retrieved Mar 30, 2023 from https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[43] Behnam Pourghassemi, Chenghao Zhang, Joo Hwan Lee, and Aparna Chandramowlishwaran. 2020. On the Limits of Parallelizing Convolutional Neural Networks on GPUs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 567–569. https://doi.org/10.1145/3350755.3400266

[44] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[45] Amir Roth and Gurindar S. Sohi. 2000. Register Integration: A Simple and Efficient Implementation of Squash Reuse. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (Monterey, California, USA) *(MICRO 33)*. Association for Computing Machinery, New York, NY, USA, 223–234. https://doi.org/10.1145/360128.360151

[46] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 194–205. https://doi.org/10.1145/264107.264200

[47] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[48] Kai Wang and Calvin Lin. 2017. Decoupled Affine Computation for SIMT GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/3079856.3080205

[49] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 47–61. https://doi.org/10.1145/3037697.3037729

[50] S.J.E. Wilton and N.P. Jouppi. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (1996), 677–688. https://doi.org/10.1109/4.509850

[51] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. 2013. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 433–442. https://doi.org/10.1145/2464996.2465022

[52] Yi Yang, Ping Xiang, Michael Mantor, Norman Rubin, Lisa Hsu, Qunfeng Dong, and Huiyang Zhou. 2014. A Case for a Flexible Scalar Unit in SIMT Architecture. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, USA, 93–102. https://doi.org/10.1109/IPDPS.2014.21

[53] Tsung Tai Yeh, Roland N. Green, and Timothy G. Rogers. 2020. Dimensionality-Aware Redundant SIMT Instruction Elimination. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1327–1340. https://doi.org/10.1145/3373376.3378520

[54] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. 2015. A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-Level Parallelism in GPGPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/2751205.2751234