Instruction Roofline: An Insightful Visual Performance Model for GPUs

Nan Ding*[1] | Muaaz Awan[2] | Samuel Williams[1]

[1]Computational Research Division, Lawrence Berkeley National Laboratory, CA, USA
[2]National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, CA, USA

Correspondence
*Nan Ding Email: nanding@lbl.gov

Present Address
1 Cyclotron Road, Berkeley, CA 94720, USA

Summary

The Roofline performance model provides an intuitive approach to identify per-formance bottlenecks and guide performance optimization. However, the classic FLOP-centric approach is inappropriate for the emerging applications that perform more integer operations than floating point operations. In this paper, we reintro-duce our Instruction Roofline Model on NVIDIA GPUs and expand our evaluation of it. The Instruction Roofline incorporates instructions and memory transactions across all memory hierarchies together, and provides more performance insights than the FLOP-oriented Roofline Model, i.e., instruction throughput, stride mem-ory access patterns, bank conflicts, and thread predication. We use our Instruction Roofline methodology to analyze eight proxy applications: HPGMG from AMReX, Matrix Transpose benchmarks, ADEPT from MetaHipMer's sequence alignment phase, EXTENSION from MetaHipMer's local assembly phase, CUSP, cuSPARSE, cudaTensorCoreGemm, and cuBLAS. We demonstrate the ability of our methodol-ogy to understand various aspects of performance and performance bottlenecks on NVIDIA GPUs and motivate code optimizations.

KEYWORDS:
Instruction Roofline Model, NVIDIA GPUs, memory patterns

1 | INTRODUCTION

Migrating an application to a new architecture is a challenge, not only in porting the code but also in understanding and tuning the performance. Rather than manually performing the analysis, developers tend to use tools to motivate the optimization. Therefore, performance analysis tools are becoming one of the most critical components for modern architectures. Performance modeling, the critical technology to quantify performance characteristics and identify potential performance bottlenecks associated with machine capabilities, is becoming an indispensable tool for understanding performance behavior and guiding performance optimization.

The Roofline model[1] is a visually-intuitive method for users to understand performance by coupling together floating-point performance, data locality (arithmetic intensity), and memory performance into a two-dimensional graph. The Roofline model[2,3,4] can tell whether the code is either memory-bound across the full memory hierarchy or compute-bound. Unfortunately, even with sufficient data locality, one cannot guarantee high performance. Many applications perform more integer operations than floating-point, and there are applications in emerging domains, e.g., graph analytics, genomics, etc.. that perform no floating-point operations at all. The classic, FLOP-centric Roofline model is inappropriate for such domains. To that end, we develop an Instruction Roofline Model for GPUs to affect performance analysis of integer-heavy computations.

This work is a substantial expansion of our previous work[5] which introduced the instruction Roofline model for NVIDIA GPUs. Here, we include a more detailed explanation of our methodology as well as its applicability to four new kernels. Specifically, the new contributions of this paper include:

- A more detailed discussion of the effects of thread predication on performance, instruction throughput, and instruction intensity.
- In addition to the five benchmarks (HPGMG, BatchSW, Matrix Transpose, and cudaTensorCoreGemm and cuBLAS) in our original PMBS paper, in this paper, we expand our evaluation of the Instruction Roofline Model by including four additional benchmarks: ADEPT from MetaHipMer's sequence alignment phase (integer-only), EXTENSION from MetaHipMer's local assembly phase (hash-table, integer-only), CUSP (Sparse-Matrix-Vector-Multiply, spmv_csr_vector_kernel template), and cuSPARSE (csrmv API).
- A summary table that provides a comprehensive list of the metrics extracted from NVIDIA's Nvprof and Nsight Compute tools required to construct and Instruction Roofline Model.
- The Instruction Roofline model profiling and plotting scripts[6] including GIPS, Instruction Intensity and memory walls. With the metric table above, these scripts, and access to an NVIDIA GPU, readers should be able to apply our methodology and analysis to their own kernels or applications.

2 | THE CLASSIC ROOFLINE MODEL

The Roofline model characterizes a kernel's performance in GigaFLOPs per second (GFLOP/s) as a function of its arithmetic intensity (AI), as described as Eq.(1). The AI is expressed as the ratio of floating-point operations performed to data movement (FLOPs/Bytes). For a given kernel, we can find a point on the X-axis based on its AI. The Y-axis represents the measured GFLOP/s. This performance number can be compared against the bounds set by the peak compute performance (Peak GFLOP/s) and the memory bandwidth of the system (Peak GB/s) to determine what is limiting performance: memory or compute.

$$\text{GFLOP/s} \leq \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Peak GB/s} \times \text{Arithmetic Intensity} \end{cases} \tag{1}$$

The classic Roofline model has been successfully used for performance analysis on different architectures. In prior work, researchers created additional compute ceilings (e.g. "no FMA" peak) and memory ceilings (e.g., cache levels)[7,3,8]. However, such refinement is misplaced when the bottleneck is not floating-point in nature but pertains to integer instruction throughput or memory access.

3 | INSTRUCTION ROOFLINE MODEL FOR GPUS

Intel Advisor[9] introduced the ability to analyze integer-heavy applications and generate scalar and vector integer *operation* (IntOP) ceilings. At its core, the Roofline model in Intel Advisor is based on *operations* (floating-point or integer). In either case, the vertical axis remains performance (FLOP/s, *int*/s, or *int/s+FLOP/s*) while the horizontal axis remains Arithmetic Intensity (operations per byte). Whereas Intel Advisor can be quite effective in this regard, it suffers from two aspects. First, it only captures pipeline throughput and may not detect instruction fetch-decode-issue bottlenecks. Second, it is an Intel-only solution. The latter is particularly troublesome given the ascendancy and vendor-breadth of accelerated computing.

In order to affect the Instruction Roofline analysis for GPU-accelerated applications, we need to target a different set of metrics. First, rather than counting floating-point and/or integer operations, we count instructions. Counting instructions allows us to both identify fetch-decode-issue bottlenecks, and, when categorized by types, pipeline utilization. Thread predication is another critical performance factor on GPUs. When a branch is executed, threads that don't take the branch are predicated (masked in vector parlance) so that they do not execute subsequent operations. When predication is frequent, one may observe poor kernel performance as very few threads execute work on any given cycle. Finally, the nature of GPU computing makes efficient data movement a critical factor in application execution time. As such, it is essential that we also characterize the global and shared memory access patterns to assess the efficiency of data motion and motivate future code optimization. Although developers can

use nvprof[10] and nvvp[11] to diagnose the performance bottlenecks discussed above, the Instruction Roofline Model provides an approachable means of characterizing performance bottlenecks in a single figure.

3.1 | Architectural Characterization

First, we describe how we define the Instruction Roofline ceilings and memory pattern walls. Here we use NVIDIA's V100 GPU (GV100)[12] to describe the methodology, but it is applicable to any GPU architecture.

Instructions and Bandwidth Ceilings: Each GV100 Streaming Multiprocessor (SM) consists of four processing blocks (warp schedulers), and each warp scheduler can dispatch one instruction per cycle. As such, the theoretical maximum (warp-based) instruction/s is $80(\text{SM}) \times 4(\text{warp scheduler}) \times 1(\text{instruction/cycle}) \times 1.53(\text{GHz}) = 489.6$ GIPS. Memory access is coalesced into transactions. The transaction size for global/local memory, the L2 cache, and HBM are 32 bytes. The shared memory transaction size is 128 bytes. In practice, a warp-level load may generate anywhere from 1 to 32 transactions depending on memory patterns. This makes the "transaction" the natural unit when analyzing memory access. We leverage Yang et al.'s methodology[2] for measuring GPU bandwidths but rescale into billions of transactions per second (GTXN/s) based on the transaction size.

The Instruction Roofline Model is described in Eq.(2). A kernel's performance, characterized in billions of instructions per second (GIPS), is a function of peak machine bandwidth (GTXN/s), Instruction Intensity, and machine peak GIPS. "Instruction Intensity" on the GPU is defined as warp-based instructions per transaction. Figure 1 shows the resultant Instruction Roofline ceilings for the GV100. L1, L2, and HBM bandwidths of 14,000, 2,996, and 828 GB/s sustain 437, 93.6, and 25.9 GTXN/s when normalized to the typical 32-byte transaction size.

$$\text{GIPS} \leq \min \begin{cases} \text{Peak GIPS} \\ \text{Peak GTXN/s} \times \text{Instruction Intensity} \end{cases} \tag{2}$$

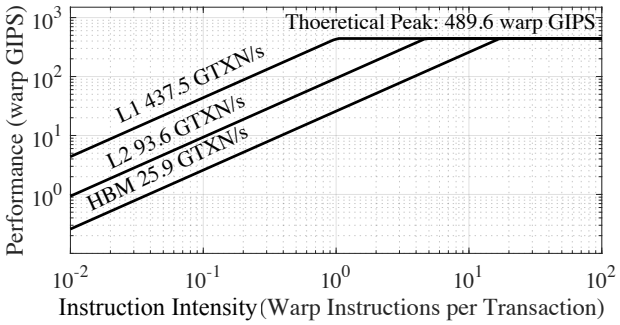


FIGURE 1 Instruction Roofline Model for the GV100. 1 GTXN/s is 10^9 transactions per second.

Global Memory Walls: When a warp executes an instruction to access global memory, it is crucial to consider the access pattern of threads in that warp because inefficient memory access can lower the performance by generating superfluous transactions.

As discussed, a warp-level load instruction can generate anywhere from 1 to 32 transactions. We can recast such a ratio into two key instruction intensities: 1 and $1/32$ warp-level *global* loads per *global* transaction. The former arises when all threads in a warp reference the same memory location and only a single transaction is generated. We call this "stride-0". Conversely, the other extreme can occur for a number of scenarios including random access, and striding by over 32 bytes ("stride-8" if FP32/INT32 and "stride-4" if FP64). Unit-stride ("stride-1"), memory access provides a global Load/Store intensity of $1/8$ (FP64) and $1/4$ (FP32, INT32). Thus, on the Instruction Roofline, we may plot three intensity "walls" representing stride-0, stride-1 (unit-stride), and stride-8.

Shared Memory Walls: GPU shared memory is a highly-banked structure within each SM providing 4-byte access. In theory, this allows for all 32 threads in a warp to make concurrent random access to shared memory in a single 128-byte transaction. However, as there are only 32 banks on the GV100, two threads contending for the same bank but different 4-byte words will cause a "bank conflict" and multiple transactions will be generated. In the worst case, all 32 threads hit different 4-byte words

in the same bank, and 32 transactions are generated. As with global/local memory walls, we can visualize bank conflicts on the Instruction Roofline Model. Note, there are two key instruction intensities: 1 and 1/32 warp-level *shared* loads per *shared* transaction.

3.2 | Application Characterization

Mirroring the previous section that characterizes GPU performance capabilities, in this section, we describe the methodology we employ to characterize application execution in terms of the Instruction Roofline Model.

Instruction Intensity and Performance: The instruction Roofline requires we measure three terms. We use `inst_executed_thread/32` to record the number of instructions executed by each kernel (scaled to warp-level), and `nvprof -print-gpu-summary` to extract kernel run time. We use the sum of `gld_transactions` and `gst_transactions` to record the total number of global transactions (for L1) and the sum of `shared_load_transactions` and `shared_store_transactions` to record the total number of shared transactions (for L1). Similarly, we use the sum of `l2_read_transactions` and `l2_write_transactions` to record the total number of L2 transactions, and the sum of `dram_read_transactions` and `dram_write_transactions` to record the total number of HBM transactions. The ratio $\frac{\text{inst\_executed\_thread}/32}{\text{HBM transactions}}$ is the HBM Instruction Intensity while $\frac{\text{inst\_executed\_thread}/32}{1e9 * \text{run time}}$ is instruction performance in GIPS.

Figure 2 visualizes the resultant Instruction Roofline Model for an arbitrary kernel. As with the traditional Roofline, one may infer cache reuse based on the distance between points (no reuse) and performance bounds based on how closes each colored point is to the associated colored bandwidth ceiling or peak performance.

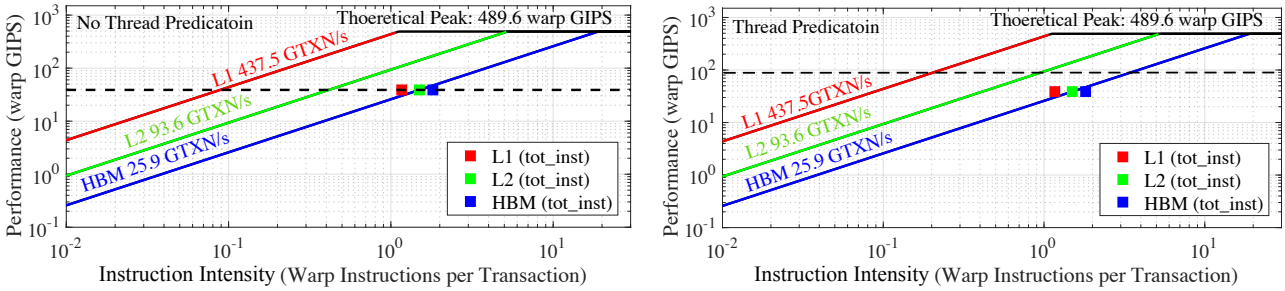
Tensor Cores: Tensor Cores in GV100 are programmable matrix-multiply-and-accumulate units that can deliver up to 125 TFLOP/s. Whereas the traditional Roofline is premised on "operations", as discussed in the previous section, we can recast intensity in terms of instructions. We can take that one step further and use floating-point instructions or tensor instructions. Such metrics are particularly useful in understanding why pipeline utilization can be high even if FLOP/s is low. We use `smsp_inst_executed_pipe_tensor.sum` in NSight Compute [13] to collect the total number of warp-based instructions (HMMA) executed on tensor cores. Similar to how previous work would plot floating-point performance relative to peak and a "no FMA" peak [2], we plot sustained HMMA GIPS relative to either peak GIPS or peak HMMA GIPS.

Thread Predication: Quantifying the performance impact of thread predication relative to bandwidth bottlenecks can be critical in determining the overall kernel performance impact. Moreover, thread predication is an intuitive way to understand resource utilization — the fraction of threads in one warp that are active during the execution. Recall that regardless of whether one thread in a warp is executing an instruction, or 32 threads in a warp are executing 32 instructions, only one warp-level instruction is executed. Therefore, the ratio of warp-based instructions (`inst_executed`) to thread-based instructions (inst_thread_executed/32) is the degree of predication. When the number is close to 1.0, there is little predication, and the performance impact is small.

The presence of thread predication can substantially change instruction intensity and performance. Nominally, thread predication reduces the number of instructions executed without changing run time. Combined, these tend to decrease both instruction intensity and performance at the same rate (dots move diagonally down and to the left). However, in some cases, thread predication on loads and stores can reduce the number of memory transactions. This tends to increase instruction intensity (dots move to the right).

Figure 2(a) and (b) highlight two cases of thread predication. In both figures, the dotted line represents the observed warp-level instruction performance. A dotted line close to the theoretical peak indicates instruction issue rates are bottlenecked by hardware issue rates. Thread-level instruction performance is constrained to be less than or equal to this bound (ceiling). Conversely, the dots represent thread instruction throughput normalized by 32 (warp size) for each level of memory. By definition, (normalized) thread-level instruction rates must be less than warp instruction rates. In Figure 2(a), the dots (thread-level instruction throughput) fall on the dotted line (warp-level instruction throughput) indicating no thread predication. Conversely, in Figure 2(b), the dots are well below the dotted line indicating a 2× loss in performance due to thread predication.

Global Memory Pattern Walls: Figure 3 shows the Instruction Roofline Model with Global Memory Walls. The solid red dot's instruction intensity is based on total instructions and L1 transactions that include global, local, and shared memory. Whereas global/local transactions are 32 bytes, shared memory transactions are 128 bytes. Thus, in order to calculate the number of equivalent 32-byte transactions and create a common denominator, one must scale the number of shared memory transactions by four. The resultant denominator, $1\times$global transactions $+ 4\times$shared transactions, allows for direct comparisons to the L1 ceiling and allows us to determine whether the combined effect of global, local, and shared transactions has made the code L1-bound.



(a) Instruction Roofline with no thread predication.

(b) Instruction Roofline with thread predication.

FIGURE 2 Thread predication in the context of the Instruction Roofline Model. A kernel without predication (left) has thread instruction throughput (dots) matched to the warp instruction throughput (dotted line) while a kernel with moderate predication (right) has thread instruction throughput substantially below the warp instruction throughput.

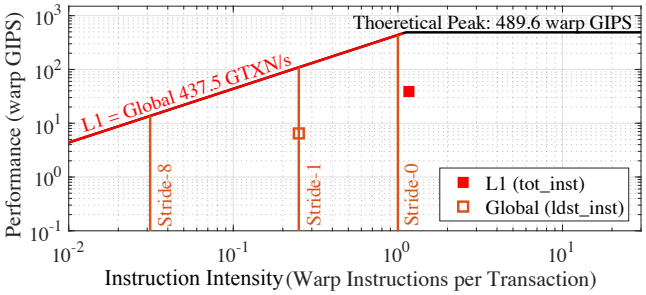


FIGURE 3 Global memory walls on the Instruction Roofline plot for the GV100. A generic kernel has been plotted on the Roofline for pedagogical purposes. The position of the open dot visualizes the stride of the memory access pattern.

We may refine it (open orange dot) to include only warp-level global load/store instructions (e.g. `inst_executed_global_loads`) and transactions (e.g. `gld_transactions`). This resultant dot will have both a different performance (GIPS) and a different instruction intensity as there are fewer load and store instructions than total instructions. The distance between the open and solid points is the fraction of load/store instructions constitute the dynamic instruction mix (close points indicate code that are mostly load/store).

The position of the load/store instruction intensity relative to the memory walls visualizes the average memory access pattern. In this generic example, we see that the dot lies on the unit-stride wall indicating the kernel accesses memory in a unit-stride manner. If it were to move to the left, one would conclude strided or gather memory access while if it were to move to the right, one would conclude multiple threads access the same word in memory. One should notice that we only count non-predicated load/store instructions here so that both load/store instructions and transactions are executed by active threads. Hence, the open dot of global memory access pattern can fall to the right of Stride-0 wall if predicated off load/store instructions are also counted. Such situation may happen when using Nsight compute to do the profiling, e.g., use `smsp__inst_executed_op_shared_ld.sum` instead of `smsp__inst_executed_op_shared_ld_pred_on_any.sum`, because the former one includes both predicated-on threads and predicated-off threads instructions.

Shared Memory Walls: Recasting our previous discussion from global memory and the L1 cache to shared memory, Figure 4 shows the Instruction Roofline Model with shared memory bandwidth and shared Memory Walls. Note, shared memory GTXN/s is less than global memory (L1 cache) GTXN/s in as the shared memory transaction size is 128 bytes while the global memory size is 32 bytes. Here, striding effects have been juxtaposed with bank conflicts. We measure the number of warp-level shared load or store instructions a kernel executes with `inst_executed_shared_loads` and `inst_executed_shared_stores` and the number of shared load and store transactions with `shared_load_transactions` and `shared_store_transactions`. We may use these terms to calculated shared memory instruction intensity.

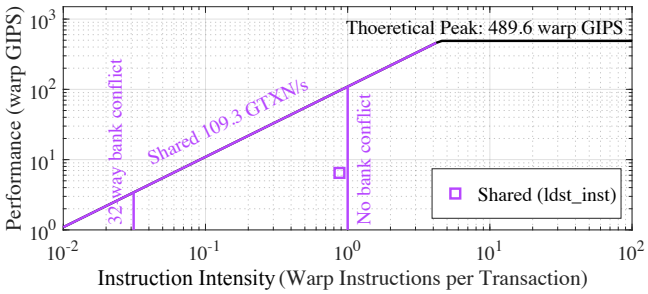


FIGURE 4 Shared memory walls on the Instruction Roofline plot on GV100. A generic kernel has been plotted on the Roofline for pedagogical purposes. The position of the open dot visualizes the number and impact of bank conflicts.

Consider the exemplar kernel in Figure 4. We may plot its shared memory instruction intensity and performance (open dot) relative to the shared memory bandwidth and walls. We observe that the kernel is efficiently accessing shared memory as it is close to the "no bank conflict" wall. Conversely, kernels that generate large numbers of bank conflicts will move to the left towards the "32-way bank conflict" wall. Similarly, we can compare the shared open dot to the shared memory ceiling to tell whether the code is bound by the shared memory.

TABLE 1 Metrics for Instruction Roofline Model

	Nvprof Metrics	**NSight Compute Metrics**	**Descriptions**
thread-based	inst_thread_executed	smsp__thread_inst_executed.sum	non-predicated
warp-based	inst_executed	smsp__inst_executed.sum	total instructions
	inst_executed_global_loads	smsp__inst_executed_op_global_ld.sum	L1 Cache Instructions
	inst_executed_global_stores	smsp__inst_executed_op_global_st.sum	
	inst_executed_local_loads	smsp__inst_executed_op_local_ld.sum	
	inst_executed_local_stores	smsp__inst_executed_op_local_st.sum	
	inst_executed_shared_loads	smsp__inst_executed_op_shared_ld.sum	
	inst_executed_shared_stores	smsp__inst_executed_op_shared_st.sum	
	gld_transactions	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum	L1 Cache Transactions
	gst_transactions	l1tex__t_sectors_pipe_lsu_mem_global_op_st.sum	
	local_load_transactions	l1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum	
	local_store_transactions	l1tex__t_sectors_pipe_lsu_mem_local_op_st.sum	
	shared_load_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum	
	shared_store_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum	
	l2_read_transactions	lts__t_sectors_op_read.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum	L2 Cache
	l2_write_transactions	lts__t_sectors_op_write.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum	
	dram_read_transactions	dram__sectors_read.sum	HBM Memory
	dram_write_transactions	dram__sectors_write.sum	
	sysmem_read_transactions	lts__t_sectors_aperture_sysmem_op_read.sum	PCIe/NVLINK
	sysmem_write_transactions	lts__t_sectors_aperture_sysmem_op_write.sum	
	-	smsp_inst_executed_pipe_tensor.sum	tensor core HMMA instruction
kernel-based	nvprof –print-gpu-summary	smsp__cycles_elapsed.sum / smsp__cycles_elapsed.sum.per_second	execution time

Summary of Metrics and Plotting: Table 1 lists the nvprof metrics used to measure instructions and data movement on GV100 CUDA core and NSight Compute metrics for tensor core HMMA instructions in this paper. Alternatively, one can choose to use NSight Compute metrics for CUDA core profiling, and the corresponding metrics are also listed.

The Instruction Roofline model profiling and plotting scripts[6] contains:

- Two scripts for profiling: *collect_metrices.sh* for cuda cores and *tensor_collect_metric.sh* for HMMA instructions on tensor cores.
- A script for plotting architecture ceilings: *irf_ceilings.m*. It plots peak GIPS and peak memory bandwidth across memory hierarchies.

- A script for plotting kernels GIPS, II, global memory walls and patterns: *global_walls.m*. It plots the GIPS and II across memory hierarchies, and the three global memory walls and patterns.
- A script for plotting kernel's shared memory walls and patterns: *shared_walls.m*.

4 | RESULTS

In this section, we describe our test machine and profiling tool, and use the Instruction Roofline Model to evaluate and analyze several GPU-accelerated applications.

4.1 | Experimental Setup

Results presented in this paper were obtained on the GPU-accelerated partition on Cori (Cori-GPU) at NERSC. Cori-GPU is comprised of nodes with two Intel Skylake CPUs and eight NVIDIA V100 GPUs, while each compute node on Summit contains two IBM POWER9 processors and six NVIDIA V100 accelerators. Nevertheless, in all experiments, we use only a single process running on one GPU and thus mitigate NVLink, PCIe, and host processor performance. We use CUDA 11 for ADEPT and EXTENSION, and CUDA 10 for the rest of the kernels, `nvprof` [10], and NSight Compute [13].

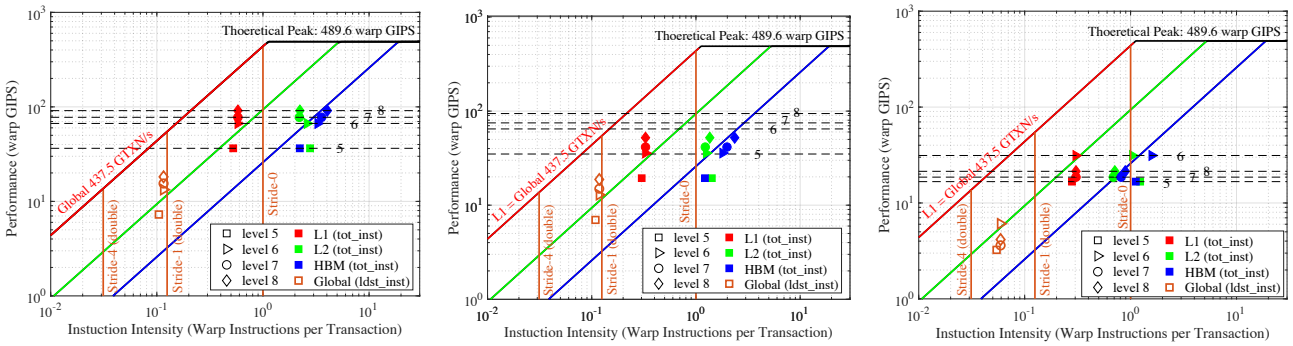
We evaluate the Instruction Roofline methodology for NVIDIA GPUs using eight proxy applications: HPGMG [14,15,16] from AMRex [17], Matrix Transpose benchmarks [18],

ADEPT [19] from MetaHipMer's [20] sequence alignment phase, EXTENSION from MetaHipMer's local assembly phase, CUSP [21] (spmv_csr_vector_kernel template) and cuSPARSE [22] (csrmv API) for sparse matrix-vector multiplication (SpMV), cudaTensorCoreGemm [23], and cuBLAS [24]. These applications exhibit a range of computational characteristics including data types, data locality, and thread predication properties. Specifically, HPGMG executes roughly 50% integer instructions and 30% floating-point instructions, matrix transpose performs almost entirely load/store instructions, ADEPT and EXTENSION performs entirely integer instructions, and both CUSP and cuSPARSE perform SpMV but show different performance, and both cudaTensorCoreGemm and cuBLAS perform HMMA instructions but use different implementations. We show that unlike the classical FLOP-centric Roofline model, the Instruction Roofline Model and our methodology can effectively analyze such a wide range of applications.

4.2 | HPGMG

HPGMG is a geometric multigrid benchmark to proxy the multigrid solves in block structured AMR (Adaptive Mesh Refinement) applications that use the AMReX framework. HPGMG solves the 4^{th} order, variable-coefficient Laplacian on a unit-cube with Dirichlet boundary conditions using a multigrid F-cycle. The Gauss-Seidel, Red-Black (GSRB) smoother dominates HPGMG's run time. GSRB smoothers perform two stencil kernel invocations per smooth (red and black). Cells are marked as either red or black in a 3D checkerboard pattern. Cells matching the sweep color are updated, while the others are copied to the result array.

HPGMG includes three different implementations of its GSRB smoother: GSRB_FP, GSRB_BRANCH, and GSRB_STRIDE2. All three perform the same computation and touch the same data over the course of a thread block's execution, but vary memory access and predication. As such, they are ideal cases to demonstrate the subtle performance differences using the Instruction Roofline Model. GSRB_BRANCH is conceptually the simplest implementation. In it, a thread block operates on a 2D slice or a 3D cache block. Within the slice, red-black execution is affected through a branch. This branch reduces the number of non-predicated threads without reducing the number of warps. The branch is eliminated in GSRB_FP through multiplication by a precomputed array of 1's and 0's. As such, predication is eliminated at the cost of doubling nominal computation whilst maintaining the same number of warp-level instructions. GSRB_STRIDE2 is similar to GSRB_BRANCH with the caveat that the 2D thread block's x-dimension is half the 2D tile's x-dimension. Thus, each thread is responsible for updating two adjacent points. Through clever address calculation, FLOPs are reduced, predication is eliminated, and the number of warp-level instructions is minimized. Note, none of these implementations use shared memory. As such, only global memory stride walls are relevant. We show how the performance insights gained from the Instruction Roofline correlate with the performance observations.



(a) Instruction Roofline for GSRB_FP. **(b)** Instruction Roofline for GSRB_BRANCH. **(c)** Instruction Roofline for GSRB_STRIDE2.

FIGURE 5 Instruction Rooflines on GV100 for the three implementations of HPGMG's GSRB. Solid dots are the total number of instructions (tot_inst) executed by non-predicated threads, open dots refer to the global memory access patterns, and dotted lines are warp-level instructions. Note, "level #" refers to the level in the multigrid v-cycle.

Figure 5 shows the Instruction Roofline on GV100 for the three implementations of GSRB as a function of multigrid level using eight 128^3 boxes on the finest level (largest arrays). On each of these figures, we overlay both total instruction intensity/performance as well as global memory access pattern (global load/store intensity/performance). The former is comparable to the nominal instruction Roofline, while the latter is shown relative to the memory stride walls. GIPS generally increases from level 5 (smallest arrays) to level 8 (largest arrays) as the overhead:surface:volume ratio improves. Figure 5a (GSRB_FP) makes it immediately obvious that: (1) memory access is unit-stride (open brown dots are very close to the stride-1 wall), (2) data reuse is captured by the L1, but not the L2 (red solid are far from green solid, but green solid are close to blue solid), and (3) the blue dots are very close to the HBM ceiling indicating GSRB_FP is HBM-bound.

Figure 5b presents the Instruction Roofline Model for GSRB_BRANCH. The thread predication impact from the branch in GSRB_BRANCH is immediately obvious in this figure as the dots (scaled thread GIPS) are well below the dashed black lines (warp-based GIPS).

Although the thread-level instruction throughput (dots) of GSRB_FP (Figure 5a) and GSRB_BRANCH (Figure 5b) differs by a factor of two, the Instruction Roofline Model for GPUs makes it clear that both implementations stress the architecture's issue bandwidth to the same degree (equal dotted lines imply equal warp-based GIPS). In terms of memory access pattern, we see no difference between GSRB_FP and GSRB_BRANCH. This should come as no surprise as the both implementations generate the same number of warp-level load/store instructions and access global memory in the same manner (same number of transactions).

Figure 5c shows the Instruction Roofline for GSRB_STRIDE2. Recall, in this implementation, there is no predication and redundant computation is minimized. Thus, the requisite number of thread and warp instructions per transaction should be further reduced which we see in reduced instruction intensity compared to Figure 5a. However, as L1 and L2 data locality crash for levels 7 and 8 (lower L2 and HBM intensity), data movement increases and the net effect is decreased performance (HBM-bound with superfluous data movement). This results in a decrease in GIPS (solid dots) with increasing level. On the converse, the Instruction Roofline shows GSRB_STRIDE2 presents a different memory access pattern from GSRB_FP or GSRB_BRANCH as its load intensity is lower than the stride-1 wall (open dots).

We use Figure 6 to further demonstrate the capability of the Instruction Roofline Model. First, we can see the execution time of level 7 and 8 in GSRB_STRIDE2 implementation is twice that of GSRB_FP and GSRB_BRANCH. This fact can be inferred from the Figure 5c and Figure 5b: GSRB_STRIDE2 and GSRB_BRANCH have a very similar number of thread instructions. As such, GSRB_STRIDE2's lower GIPS is indicative of a longer execution time. Second, GSRB_FP and GSRB_BRANCH implementations have the same execution time but very different GFLOP/s. GFLOP/s of GSRB_FP is double that of GSRB_BRANCH due to the redundant computation. The performance of levels 7-8 of GSRB_STRIDE2 is lower than GSRB_BRANCH. This is because the longer execution time of level 7-8 in GSRB_BRANCH. Resource utilization (the fraction of threads in one warp that are active during the execution) of GSRB_BRANCH is 50% while the other two implementations are 100% due to the thread predication in GSRB_BRANCH. All of these facts can be captured by the Instruction Roofline in Figure 5. The third observation in Figure 6 is that the integer instructions take nearly 50% of the total number of instructions, load/store instructions take about

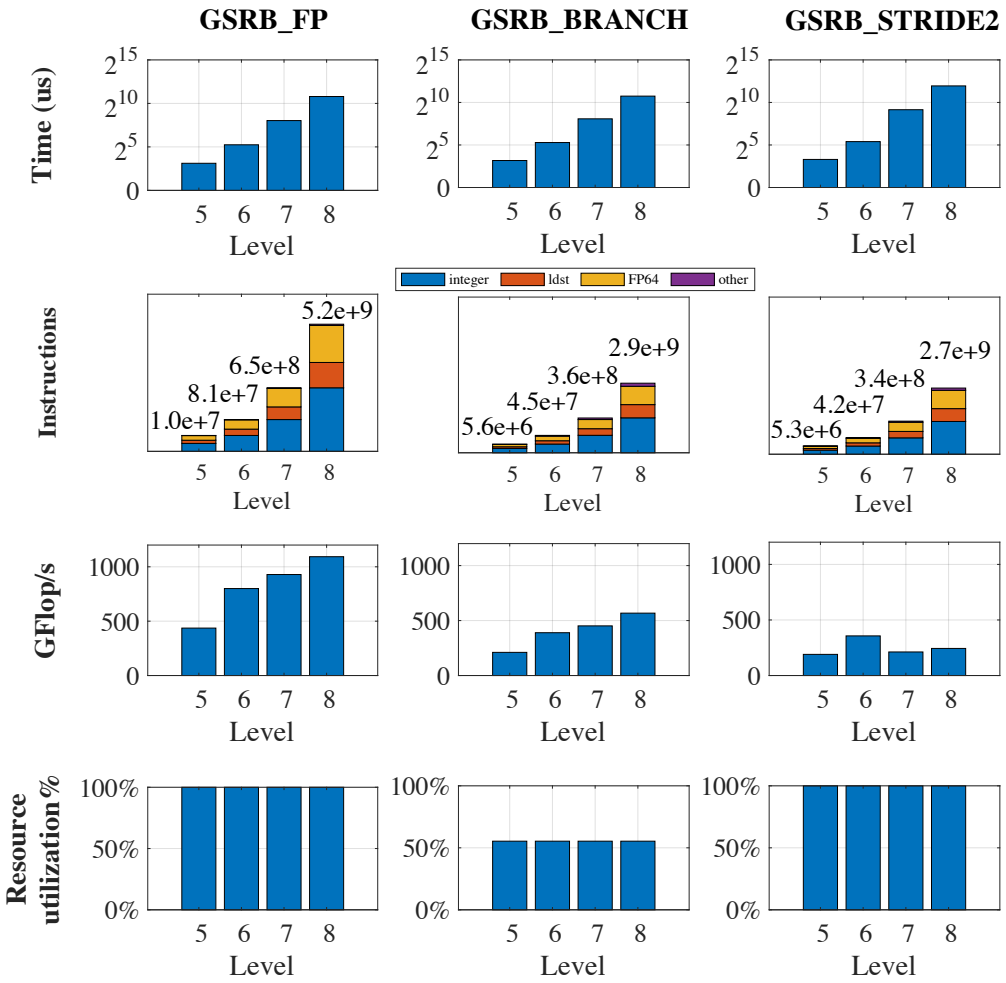


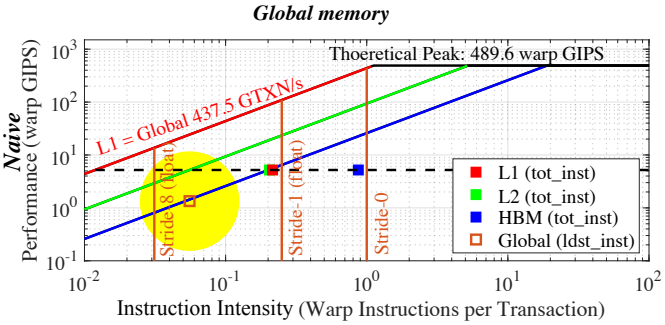
FIGURE 6 Performance insights breakdown on GV100 for the three different implementations of HPGMG as a function of multigrid level (note, level 8 represents the largest arrays). Resource utilization refers to the ratio of thread predication.

20%, and floating-point instructions take about 30%. The integer instructions are performing *IMAD*, *IADD* and *ISETP* for indexing the preparing the offsets of stencil computations. This indicates the code performs 1.6 integer instructions and 0.6 load/store instructions for every floating-point instruction. "Others" in the figure refers to the instructions like *cvt* (convert instructions), *etc.* which are not counted in any specific metric in *nvprof*.

4.3 | Matrix Transpose

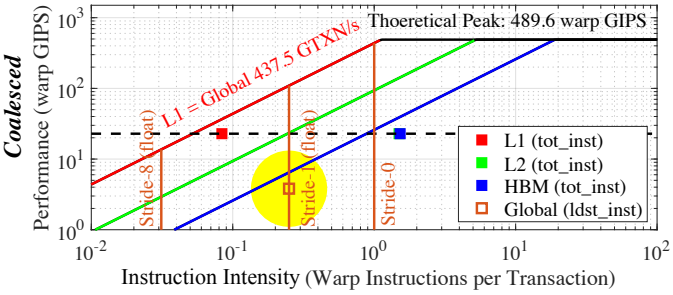
Matrix transpose flips a matrix A over its diagonal, that is, it swaps the elements of a matrix by swapping their row and column indices thereby producing a new matrix A^T. We use three different single-precision implementations (Naive, Coalesced, and Coalesced_NoBankConflict) to illustrate the different performance of both global and shared memory access patterns. In all cases, we use a 1024×1024 matrix and 32×8 thread blocks operating on 32×32 matrix tiles.

The Naive implementation uses the array index to access elements in both input arrays and output arrays. Each thread reads four elements from one column of the input matrix and writes them to their transposed locations in one row of the output matrix. As the matrices are column-major, the reads from the input matrix are coalesced while the writes to the output matrix have a stride of 4096 Bytes (1024 floats) between successive threads. This results in the worst case of 32 separate memory transactions per warp-based load/store instruction. Figure 7a plots the instruction Roofline and memory walls for the Naive implementation. Clearly, memory access (open symbol) is far from unit-stride on average. Moreover, we see poor L1 data locality but good L2

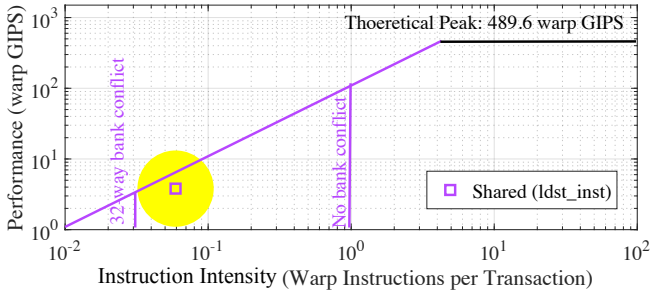


(a) Matrix Transpose Naive: global memory pattern.

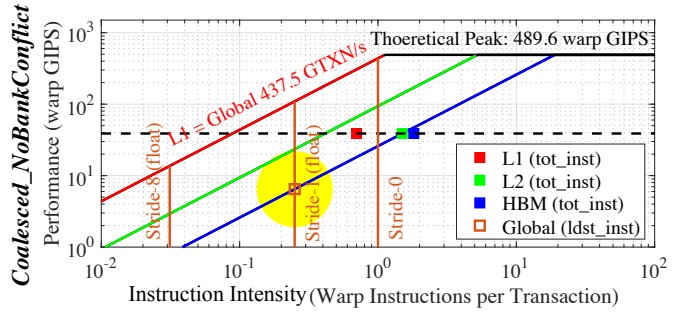
(b) Matrix Transpose Naive: shared memory pattern.



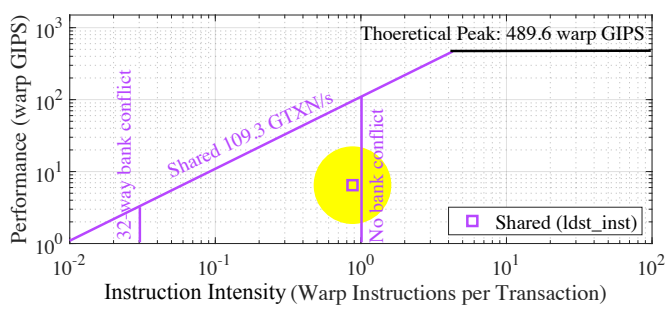
(c) Matrix Transpose Coalesced: global memory pattern.



(d) Matrix Transpose Coalesced: shared memory pattern.



(e) Matrix Transpose Coalesced_NoBankConflict: global memory pattern.



(f) Matrix Transpose Coalesced_NoBankConflict: shared memory pattern.

FIGURE 7 Instruction Roofline on GV100 for the three implementations in Matrix Transpose. The solid dots are the total number of instructions (tot_inst) executed by non-predicated threads. The open dots refer to memory access patterns.

locality (green and red dots are close but red and blue are widely separated). Note, as the Naive implementation doesn't use shared memory, there is no corresponding shared memory figure in Figure 7a.

The Coalesced implementation reads contiguous data from matrix A into rows of a shared memory buffer. After recalculating the array index, a column of the shared memory buffer is written to contiguous addresses in the matrix A^T. As such, the global memory access pattern of the Coalesced implementation is unit-stride; Figure 7c shows exactly this (open dot). We can also see from Figure 7c that that L1 cache is better utilized than the Naive implementation as the L2 intensity has moved to the right (green dot). Importantly, the number of transactions in the red solid dot (L1) is a linear combination of global and shared memory transactions. The combined effect of these has made the code nearly L1-bound (L1 solid red dot is close to the L1 ceiling).

Although the use of shared memory has substantially improved performance, Figure 7d shows this implementation produces a large number of shared memory bank conflicts — shared load intensity (open dot) is close to the "32-way bank conflict" wall. This is easily understood as the Coalesced implementation uses a 32×32 shared memory array of floats but all data in columns k and $k + 16$ are mapped to the same bank. As a result, when writing partial columns from buffer in the shared memory to rows in the output matrix, there is a 16-way bank conflict as Figure 7d shows. Concurrently, the shared intensity (open dot) is very

close to the shared memory ceiling indicating that shared transactions are dominating all L1 transactions and is the cause of the L1 bottleneck.

Proximity to the shared memory wall and ceiling can be used to motivate software optimization. The Coalesced_NoBankConflict implementation pads the shared memory array by one column $32 \times (32 + 1)$ to avoid bank conflicts. Figure 7f shows that shared load intensity has moved to the right and is now near the "no bank conflict" wall. Note, the Coalesced_NoBankConflict version has the same global load intensity (Figure 7e) as the Coalesced version (Figure 7c) because they have the same implementation for global memory access.

As a summary for Matrix Transpose, by comparing Figure 7a, Figure 7c and Figure 7e, we can see how the global memory access pattern improved from the Naive to Coalesced and Coalesced_NoBankConflict implementations. By comparing Figure 7d and Figure 7f, we can tell how the shared memory access pattern improved from the Coalesced to Coalesced_NoBankConflict implementations.

4.4 | ADEPT

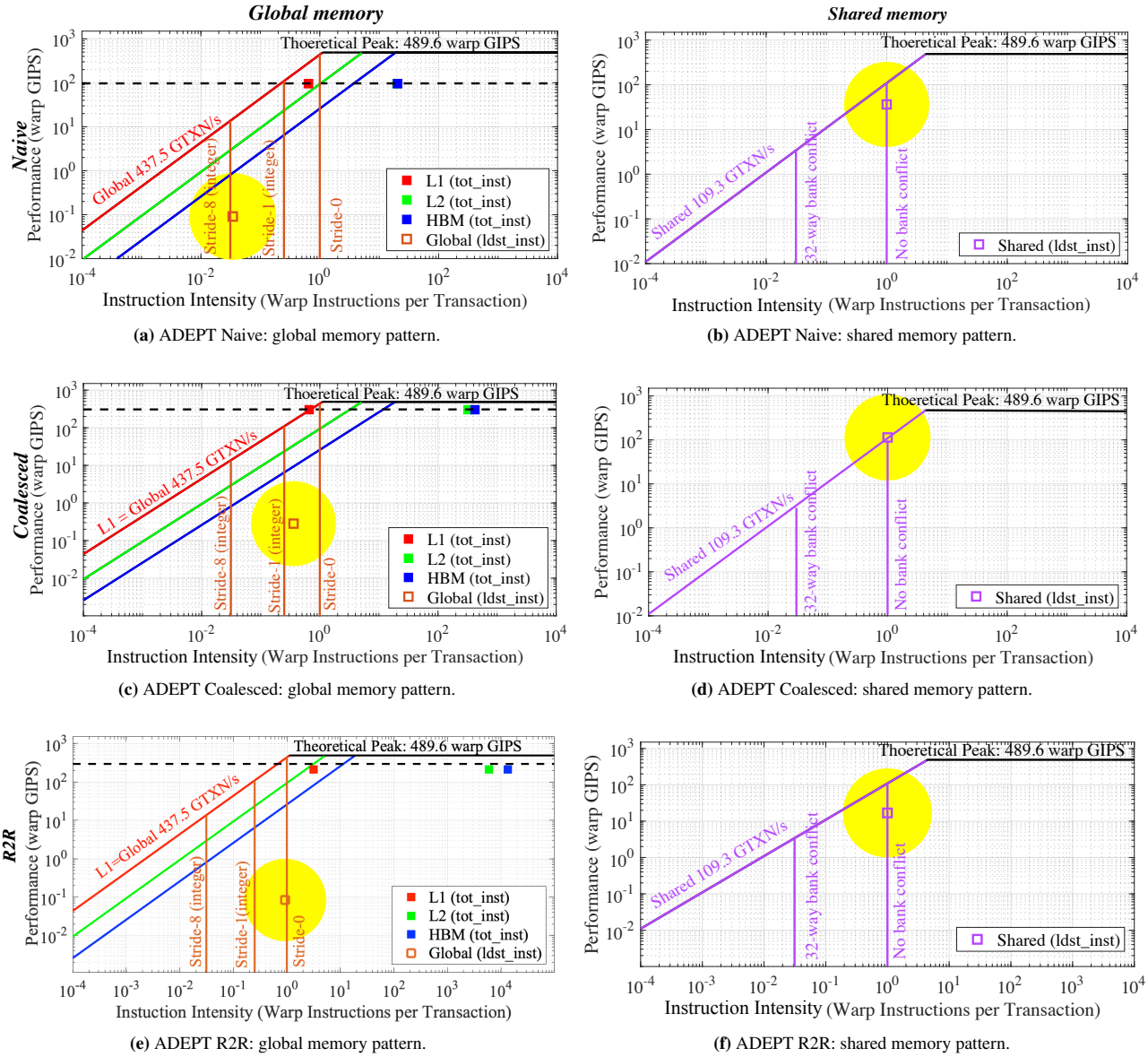
ADEPT[19] proxies MetaHipMer's sequence alignment phase[20]. MetaHipMer is a large scale *denovo* genome assembly tool that utilizes the Smith-Waterman algorithm at the core of its alignment phase. During a single node run, a significant portion of its execution time is spent in performing Smith-Waterman (SW) alignments[25]. SW is a dynamic programming algorithm that constructs an $M \times N$ matrix A given two sequences of lengths M and N.

Matrix element $A(i, j)$ is calculated as a score for aligning the i^{th} element in the first sequence with the j^{th} element in the second sequence. The score of $A(i, j)$ depends on elements $A(i, j-1)$, $A(i-1, j)$ and $A(i-1, j-1)$. Next, the optimal local alignment is defined as tracing back a path connecting the starting point (i_0, j_0) in the matrix to any point $(i, j), i > i_0, j > j_0$, with the highest sum of scores along this path. In this paper, we set $M = 128, N = 1024$ with a total number of 30,000 pairs of sequences which is representative of data processed during a single ADEPT (GPU accelerated SW) kernel call made by MetaHipMer.

From the development cycle of ADEPT we obtained three different implementations: Naive, Coalesced, and R2R. All three implementations have one kernel of two phases: scoring and tracing. The scoring phase of the two implementations is the same. The matrix A is computed in diagonal-major fashion because of the dependencies discussed above, the elements of A are computed in the sequence: $A(0,0) \rightarrow (A(1,0), A(0,1)) \rightarrow (A(2,0), A(1,1), A(0,2))$, and successive diagonals can be done in the same manner. The three most recent diagonals are stored in three different shared memory arrays in Naive and Coalesced version. Each thread in a warp writes the scores to a unique cell of the shared memory array. The R2R version stores the intermediate diagonals in thread registers instead of shared memory arrays and uses register to register transfers for communicating dependencies. The tracing phases of the three implementations are different. The Naive implementation uses a row-major data layout to store the scores for the whole matrix, whereas the Coalesced version uses a diagonal-major layout. The R2R implementation uses a reverse scoring technique for performing traceback which prevents it from storing the complete traceback matrices which are quite large in size.

Figure 8 shows the Instruction Roofline on GV100 for the three implementations. When examining total instruction throughput, it is clear the Naive implementation under performs attaining roughly 25% of the peak instruction Roofline GIPS. Comparing Figures 8a and 8c, it is immediately obvious how replacing a poor memory access pattern (every N^{th} element) in the row-major data layout with a diagonal-major data layout improved load intensity (open dots) to the point where it is bound by the stride-1 wall. Comparing Figures 8c and 8e, we show that using registers to eliminate large traceback matrices in R2R further improved load intensity to Stride-0.

Shared memory is used in the scoring phase in Naive and Coalesced implementations. Recall that the scoring phase of the two implementations is the same and are in the same kernel as the tracing phase. Each thread in one warp writes the scores to a unique cell of shared memory with the three most recent diagonals stored in shared memory. As a result, the 32 threads in a warp access 32 consecutive elements of shared memory and thus 32 different banks. Figures 8b and 8d show that both implementations attain a perfect shared intensity of 1.0 (no bank conflicts). Despite having no bank conflicts, coalesced shared load/store GIPS is very close to the shared memory roofline and thus indicative of the performance bound. The R2R version also has no bank conflicts because shared memory is only used when there is an inter-warp communication during which threads are accessing different banks of shared memory. In addition, the R2R implementation leverages registers to reduce the shared memory traffic and thus the shared load/store GIPS goes down. In fact, the R2R implementation is the fastest one among the



(a) ADEPT Naive: global memory pattern.

(b) ADEPT Naive: shared memory pattern.

(c) ADEPT Coalesced: global memory pattern.

(d) ADEPT Coalesced: shared memory pattern.

(e) ADEPT R2R: global memory pattern.

(f) ADEPT R2R: shared memory pattern.

FIGURE 8 Instruction Roofline on GV100 for the two implementations in ADEPT with visualized global memory pattern. The solid dots are the total number of instructions (tot_inst) executed by non-predicated threads. The open dots refer to the memory access patterns.

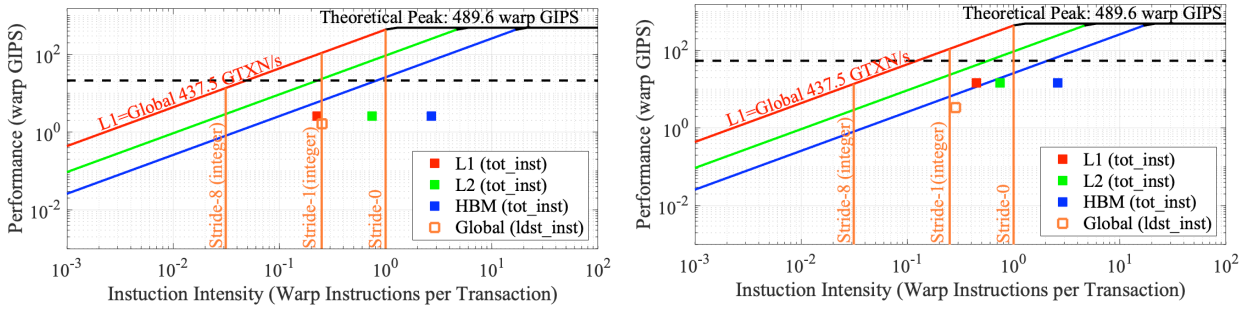
three versions, but it has a similar GIPS with the coalesced version in the Instruction Roofline plot. This is due to the combined effect of shortened kernel execution time and reduced number of instructions.

4.5 | EXTENSION Kernel

MetaHipMer's[20] local assembly phase performs local extensions on sections of DNA with the help of DNA reads. The algorithm involves two steps 1) constructing a hash table with keys as DNA sub-strings referred to here as *kmers* and values as corresponding extension characters. 2) building a DNA walk by probing the hash table for kmers[26]. We use Arctic synth data set[27] with the kmer size set to 77.

From the development cycle of GPU accelerated extension kernel we obtained the two implementations. The first uses a single CUDA thread for building the hash table as well as performing walks while the second uses a CUDA warp to build the

hash table and then uses a single thread to perform walks. We will refer to the first implementation as *thread based* and the second as *warp based*. In both implementations, the hash tables are stored in global memory. Note, as both versions don't use shared memory, there is no corresponding shared memory figures. The second step (DNA walk) is the same in both versions: one thread is assigned per hash table to perform all the probings.



(a) thread-based: instruction throughput.

(b) Warp-based: global memory pattern.

FIGURE 9 Instruction Roofline on GV100 for the EXTENSION kernel. The solid dots are the total number of instructions (tot_inst) executed by non-predicated threads. The open dots refer to the memory access patterns.

The Warp based implementation reduces the number of memory transactions for reading the DNA reads but this does not help with the memory accesses for insertion into the hash table since those will still be random owing to the nature of hash table. This behavior can be observed in Figure 9 (open dots). The global memory access pattern in both implementations are near the stride-1 wall while the open dot moves to upper right when switching from the thread based version (Figure 9a) to the warp based version (Figure 9b). It should be noted that neither kernel performs close to the peak performance. The optimized kernel (warp based) achieves 14.4 GIPS which is far from the peak value. This is mostly due to the nature of algorithm which leads to a randomly accessed global memory and large usage of local memory. Approximately 65% of L1 memory transactions are from local memory which in this case limits the performance.

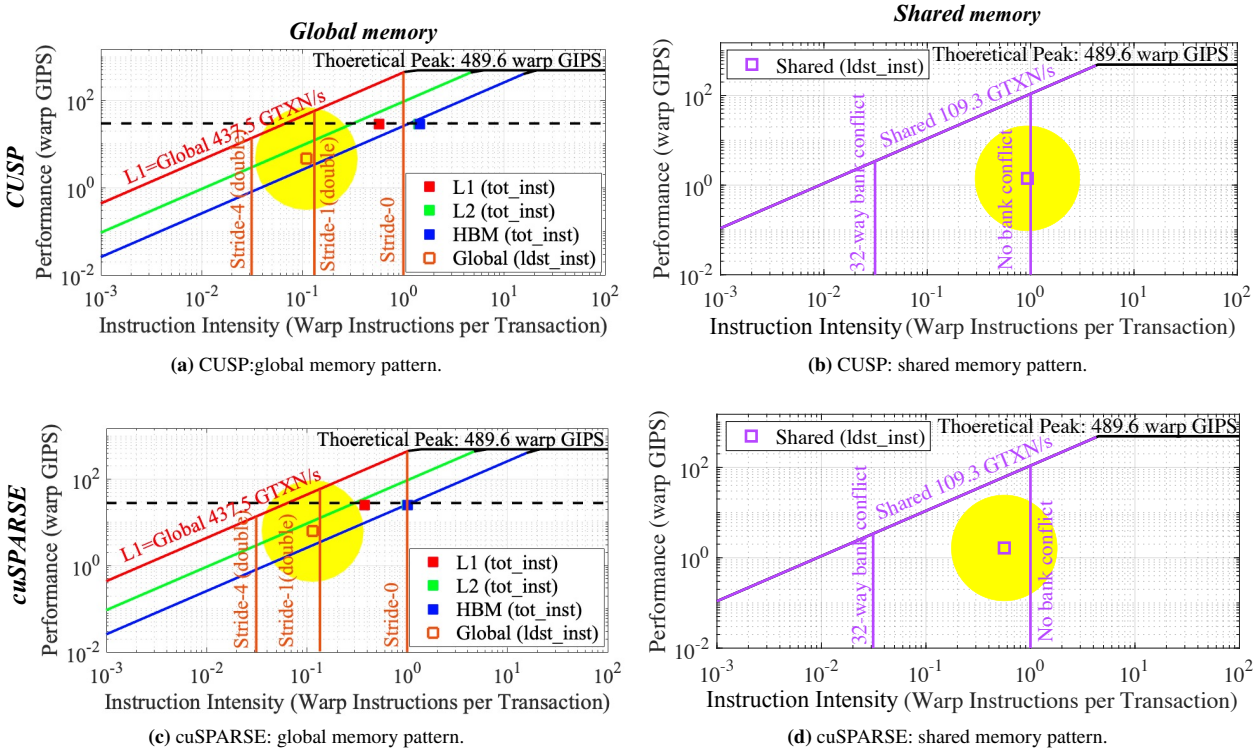
It can further be observed in Figure 9 that both kernels suffer from thread predication. This is due to the non-deterministic amount of work that each thread must perform. For instance, in the second stage of the kernel, i.e. DNA walks, some walks can be as long as 300 bases while others might terminate right at the start. This discrepancy in walk length leads to thread predication. In addition, in the walk phase, each thread might be taking a totally different path than the other. Thread predication is improved in Warp-based implementation since all 32 threads in a warp perform nearly the same amount of work and do not incur divergent branches in the first phase (build hash table). Ultimately, the EXTENSION kernel is bounded by thread prediction and high usage of local memory.

4.6 | Sparse Matrix-Vector Multiplication (SpMV)

SpMV can be denoted as $y = Ax + y$, where A is the sparse matrix, x and y are respectively the source and resultant dense vectors. Two popular GPU SpMV implementations are NVIDIA CUDA Sparse Matrix library (cuSPARSE)[22] and CUSP[21]. These two methods can have different performance when performing the same matrix-vector multiplication: cuSPARSE can attain 92 GFlops while CUSP can achieve 101 GFlops with input matrix size of 2.4 million non-zeros (4,562 rows by 5,761 columns). As such, it is important to learn the cause of the performance difference in order to drive the future code and hardware optimization. Therefore, in this section, we use our Instruction Roofline Model to evaluate these two methods.

We use *spmv_csr_vector_kernel* in CUSP to perform SpMV. In this implementation, each row of the CSR matrix is assigned to a warp. All 32 threads in one warp compute the dot product of the i-th row of A with the x vector in parallel ($y[i] = A[i, :] * x$). This work distribution implies that the CSR index and data arrays (Aj and Ax) are accessed in a contiguous manner. That is to say each warp performs 32(threads) × 8(bytes) which results in 8 global memory transactions per warp load/store instruction. This is can be observed in Figure 10a. The open dot is close to the stride-1 memory wall. Similarly, the 32 threads in one warp access a contiguous shared memory to perform row reduction so that there's no shared bank conflict in Figure 10b.

Comparing Figure 10c and Figure 10a, we observe that cuSPARSE has the same global memory pattern as CUSP. As the two implementations operate on the same matrix, by comparing Figures 10d and 10b, it is immediately obvious that the low performance of cuSPARSE is due to the 2-way shared memory bank conflicts. Therefore, one can infer that CUSP has a better shared memory access pattern than cuSPARSE, and thus, can attain higher performance. Broadly speaking, cuSPARSE and other NVIDIA developers can leverage the Instruction Roofline Model's bank conflict walls to quickly identify cases where bank conflicts are impeding performance.



(a) CUSP:global memory pattern.

(b) CUSP: shared memory pattern.

(c) cuSPARSE: global memory pattern.

(d) cuSPARSE: shared memory pattern.

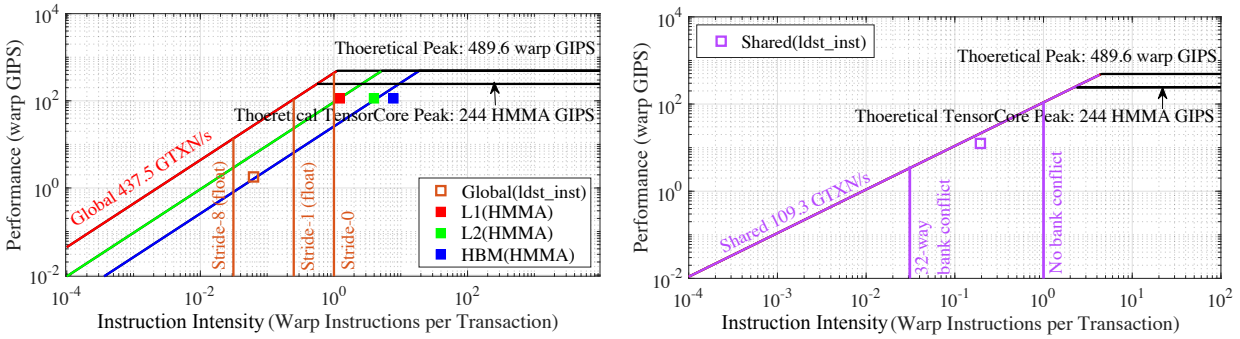
FIGURE 10 Instruction Roofline on GV100 for the SpMV. The solid dots are the total number of instructions (tot_inst) executed by non-predicated threads. The open dots refer to the memory access patterns.

4.7 | Matrix Multiplication using Tensor Cores

Each tensor core can complete a single 4×4 FP16 matrix multiplication and FP32 accumulation per cycle, i.e. $D = A \times B + C$, where A, B, C are 4×4 matrices [12]. Currently, the lowest level interface to program Tensor Cores is CUDA's Warp Matrix Multiply and Accumulation (WMMA) API [28]. The WMMA API provide warp-wide operations for performing the computation of $D = A \times B + C$, where A (FP16), B (FP16), C (FP32) and D (FP32) can be tiles of larger matrices. All threads in a warp cooperatively work together to perform a matrix-multiply and accumulate operation on these tiles using the WMMA API. Matrix A($M \times N$) and C($M \times K$) are row-major and matrix B($N \times K$) is column-major.

Another popular method to perform matrix multiplications is to use cuBLAS [29]. The cuBLAS library is an highly-tuned implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. We use the cuBLASLt API supported by CUDA 10.1. The cuBLASLt is a new lightweight library dedicated to GEneral Matrix-matrix Multiply (GEMM) operations.

Unfortunately, these two methods can have very different performance when performing the same matrix multiplications — cuBLAS can attain 104 TFLOP/s while WMMA can only attain 58.23 TFLOP/s. As such, it is essential to understand the performance nuances of these two methods in order to motivate the future code and hardware optimization. To that end, in this section we use the Instruction Roofline Model to evaluate both approaches using test matrices of size $M = N = K = 32,768$.



(a) Global memory pattern of cudaTensorCoreGemm

(b) Shared memory pattern of cudaTensorCoreGemm

FIGURE 11 Instruction Roofline on GV100 for the cudaTensorCoreGemm (WMMA interface). The solid dots are HMMA instructions while the open dots are global/shared loads/stores.

WMMA Roofline: There are three WMMA instructions in cudaTensorCoreGemm: `wmma.load`, `wmma.store`, and `wmma.mma`. `wmma.load` and `wmma.store` are broken into a group of normal load and store instructions. Each `wmma.mma` instruction is broken into 16 HMMA instructions for mixed precision. As such, there are $16 \times \frac{M \times N \times K}{16 \times 16 \times 16}$ HMMA instructions, where $16 \times 16 \times 16$ matrix operations can be performed by one WMMA instruction. In other words, there are 512 floating-point operations per HMMA instruction. Thus, we may define tensor core HMMA peak instruction ceiling by tensor core peak TFLOP/s by 512: $\frac{125 \times 1e3 \text{ GFLOP/s}}{512} = 244$ HMMA GIPS.

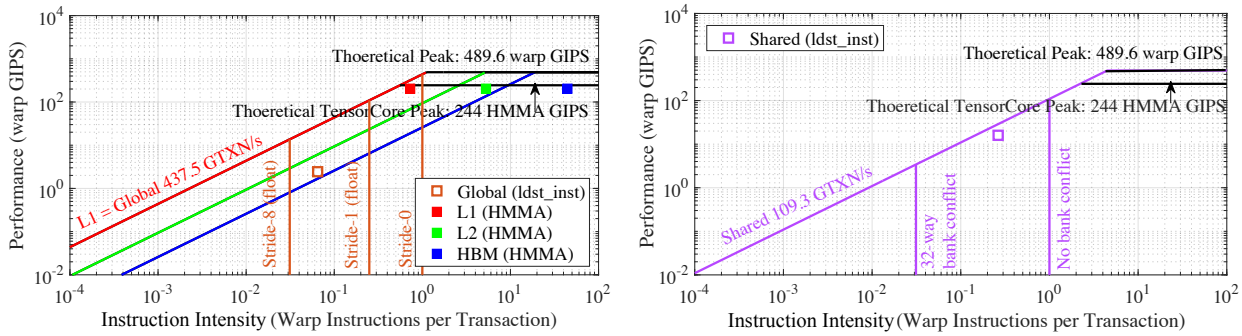
In cudaTensorCoreGemm, a 128×128 tile is computed each iteration. Within each tile, each warp computes eight 16×16 sub-tiles using `wmma.mma` operations by iterating through the full A and B matrices and accumulating the intermediate result in the local thread state. At the beginning of each iteration, a CUDA block of eight warps copies a 128×128 tile of the two input matrices from the global memory to shared memory with warps 0-3 copying matrix A and warps 4-7 copying matrix B. Each warp is assigned 64 16×16 sub-matrices and works on one at a time. Threads in one warp are organized as a 2×16 block, i.e., thread 0 loads element (0,0) to element (0,7), thread 1 loads element (1,0) to element (1,7), thread 15 loads element (0,8) to element (0,15), etc.. Thus, each warp loads the corresponding data with a stride of eight elements using 16 transactions. As such, we can see the global memory pattern in Figure 11a is between the stride-8 and the stride-1 walls.

Before performing the `wmma.mma` operation, each warp loads data using the `wmma.load`. The shared memory access pattern is not explicitly specified, but each thread in the warp can read one or multiple matrix elements from different matrix rows or columns. The worst case is that each thread accesses a different row mapped into the same bank (32-way bank conflict). As such, portions of the A and B matrices are stored in shared memory with an additional padding to reduce the number of shared memory access bank conflicts. The number of eight FP16 elements is chosen as the minimum shift in cudaTensorCoreGemm. This is because `wmma.load` requires 128-bit alignment. As a result, Figure 11b shows that bank conflicts are reduced to only 6-way. As Figure 11b also shows that the WMMA code is bound by the shared memory bandwidth, further reductions in bank conflicts can improve performance.

cuBLAS Roofline: Our cuBLAS benchmark simply calls *cublasGemmEx* to perform the matrix-to-matrix Multiply. Figure 12 shows the Instruction Roofline plot for it. Observe, cuBLAS has the same global and shared memory access pattern with cudaTensorCoreGemm (open dot). However, cuBLAS performance (solid dot) has increased with increased instruction intensity compared to cudaTensorCoreGemm in Figure 11. As the two implementations perform the same number of HMMA instructions, one can infer that cuBLAS requires fewer transactions than cudaTensorCoreGemm (better register and L2 locality) and has a higher performance as it is L1-bound.

5 | CONCLUSIONS AND FUTURE WORK

We have developed and applied a methodology based on the Roofline model for analyzing instruction throughput for kernels running on NVIDIA GPUs. This allows us to analyze both total instruction throughput (fetch-decode-issue) as well as functional unit utilization (FPU, tensor, integer, etc...) thereby expanding the applicability of Roofline to several emerging computational



(a) Global memory pattern of cuBLAS

(b) Shared memory pattern of cuBLAS

FIGURE 12 Instruction Roofline on GV100 for the cuBLAS. The solid dots are just HMMA instructions while the open dots are global/share load/store.

domains. With the insight that load/store instructions coupled with transaction categorization allows us to quantify both the memory access pattern (e.g. unit-stride vs. gather/scatter) as well as the frequency of shared memory bank conflicts, we were able to incorporate both aspects into the Roofline model as "walls" that denote the efficiency of memory access. The unified visualization of bandwidth and access efficiency endows users with far greater insights as to how different aspects of modern GPU architectures constrain performance.

In essence, both pathological memory access patterns and thread predication can substantially change instruction intensity and performance. An inefficient global memory access pattern increases the number of memory transactions per load/store instruction. More memory transactions result in longer memory latency. Combined, these tend to decrease the instruction intensity and GIPS, and make the memory access pattern dot move toward the leftmost (inefficient) memory wall. Similarly, the presence of thread predication on load/store instructions can reduce memory transactions and thus increase instruction intensity. Conversely, when thread predication is applied to all types of instructions (rather than just load/store) within a basic block, one sees a decrease in both instruction intensity and performance.

Interestingly, across benchmarks, we observe that roughly 50% of the dynamic instructions are classified by NSight compute as floating-point, integer, or load-store. This allows us to analyze each of these classes independently (e.g. traditional FLOP Roofline or understanding global memory access patterns via load/store walls). In the future, we will examine the other 50% to identify any instruction execution bottlenecks they manifest.

With GPUs having come to dominate the Department of Energy's Leadership Computing Facilities, it is only natural that much of our future work be directed towards extending our methodology to support NERSC's Perlmutter (NVIDIA A100), OLCF's Frontier (future AMD GPUs), and ALCF's Aurora (future Intel GPUs). The extension to support NVIDIA A100 from the current V100 is straightforward. Conversely, we will leverage AMD's ROCprof, Intel's Advisor, and Intel's VTune in order to extend our instruction Roofline model to AMD and Intel GPUs. Support for all three GPU architectures will allow cross-platform analysis of instruction intensity, memory walls, and instruction-class specific bottlenecks. This will allow us to quantify whether one GPU requires fewer instructions to execute a computation than another. Similarly, it allows us to understand whether some GPUs are more sensitive to memory access pattern or bank conflicts than others.

Orthogonal to cross-platform Instruction Roofline model is the ability to affect cross-programming model and cross-compiler analysis using the Instruction Roofline model. Specifically, we will use the GPU Instruction Roofline model to analyze why one programming model or compiler outperforms another.

ACKNOWLEDGEMENTS

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.This research used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which

is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.We thank NVIDIA Corporation for their willingness to answer our myriad of questions on nvprof metrics. We thank Tan Nguyen for providing his SpMV implementations and answering questions on them.

References

1. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 2009; 52(4): 65–76.
2. Yang C, Kurth T, Williams S. Hierarchical Roofline Analysis for GPUs: Accelerating Performance Optimization for the NERSC-9 Perlmutter System. *CCPE* 2019.
3. Koskela T, Matveev Z, Yang C, et al. A novel multi-level integrated Roofline model approach for performance characterization. In: Springer. ; 2018: 226–245.
4. Williams SW. *Auto-tuning performance on multicore computers*. University of California, Berkeley . 2008.
5. Ding N, Williams S. An instruction roofline model for GPUs. *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* 2019: 7–18.
6. Instruction Roofline Model Profiling and Plotting Scripts.. https://github.com/nanding0701/Instruction_roofline_scripts; .
7. Doerfler D, Deslippe J, Williams S, et al. Applying the roofline performance model to the intel xeon phi knights landing processor. In: Springer. ; 2016: 339–353.
8. Ilic A, Pratas F, Sousa L. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 2014; 13(1): 21–24.
9. Integer Roofline Modeling in Intel Advisor.. https://software.intel.com/en-us/articles/a-brief-overview-of-integer-roofline-modeling- in-intel-advisor#ref_releasenotes; .
10. Nvidia Profiler User's Guide.. https://docs.nvidia.com/cuda/profiler-users-guide/; .
11. NVIDIA Visual Profiler.. https://developer.nvidia.com/nvidia-visual-profiler; .
12. NVIDIA Tesla V100 GPU Architecture.. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf; .
13. Nsight Compute Command Line Interface.. https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf; .
14. HPGMG CUDA Code.. https://bitbucket.org/nsakharnykh/hpgmg-cuda; .
15. HPGMG Website.. https://hpgmg.org/; .
16. HPGMG-FV Documentation.. http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg; .
17. AMReX Documentation.. https://amrex-codes.github.io/amrex/; .
18. Matrix Transpose Code.. https://github.com/NVIDIA-developer-blog/code-samples/; .
19. Awan MG, Deslippe J, Buluc A, et al. ADEPT: a domain independent sequence alignment strategy for gpu architectures. *BMC bioinformatics* 2020; 21(1): 1–29.
20. Hofmeyr S, Egan R, Georganas E, et al. Terabase-scale metagenome coassembly with MetaHipMer. *Scientific reports* 2020; 10(1): 1–11.
21. CUSP: Main Page. https://cusplibrary.github.io/; . (Accessed on 02/02/2021).
22. Naumov M, Chien L, Vandermersch P, Kapasi U. Cusparse library. In: GPU Technology Conference. ; 2010.

23. CUDA Samples. http://hpc-simulations.utp.ac.pa/wp-content/uploads/2018/04/CUDA_Samples-min.pdf; .

24. Tensor Core Miniapp. https://github.com/PointKernel/tensor-core-miniapp; .

25. Zhao M, Lee WP, Garrison EP, Marth GT. SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications. *PloS one* 2013; 8(12): e82138.

26. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. meraligner: A fully parallel sequence aligner. In: IEEE. ; 2015: 561–570.

27. Hofmeyr S, Egan R, Georganas E, et al. Terabase-scale metagenome coassembly with metahipmer. *Scientific reports* 2020; 10(1): 1–11.

28. CUDA C Programming Guide (CUDA 9.0).. https://docs.nvidia.com/cuda/archive/9.0/cuda-c-programming-guide/; .

29. cuBLAS. https://docs.nvidia.com/cuda/cublas/; .