Tensile: Auto-tuning GEMM GPU Assembly for All Problem Sizes

David E. Tanner
Advanced Micro Devices

Abstract—**Generic matrix matrix multiplication (GEMM) on graphics processors (GPU) has long been the target of both tuning to find a fastest kernel for a GPU and hand-written assembly to achieve highest possible efficiency. Tensile is an open source (https://github.com/ROCmSoftwarePlatform/Tensile) tool which builds on both of these, by auto generating assembly kernels, auto benchmarking the kernels against a range of problem sizes, then auto generating library C code which selects the optimal kernel for a given problem size. Not only can the kernels achieve near-peak efficiency for the best problem sizes, but the size-tuned kernels can achieve speedups of 2X to 350X compared to any single-tuned kernel optimized for a given GPU architecture.**

I. Introduction

While generic matrix-matrix multiplication (GEMM) has always been fundamental to scientific and high-performance computing [1], the growing use of neural networks, such as convolutional [2] and recurrent [3], adds renewed demand for fast GEMM implementations [4].

GEMM's unique O(N^3) compute complexity vs O(N^2) memory complexity, make it an ideal candidate for graphics processing unit (GPU) acceleration with very high efficiency [5]. Also, the high performance per Watt and performance per dollar of GPUs, versus their CPU counterpart, make then an economical tool for scientific and industrial computing [6], [7].

However, the GPU's unique architecture require highly tuned kernels in order to achieve the peak throughput they offer [8]. The continual evolution of GPU architecture, its widening adoption into more diverse fields of research and the continual push to higher efficiency drive the need to perform ongoing tuning of GEMM on GPUs [9].

II. Previous Research

Much has been accomplished for designing ideal GEMM kernels for recent GPU architectures by using batching [10], improving auto-tuning techniques [11], [12], and utilizing caches [9].

GEMM kernel optimization efforts are generally carried out in high level languages [13] or in assembly [14]. The benefit of source languages is portability across hardware vendors and architecture generation, relying on the vendors' compilers to convert highly optimized source code into equally optimized ISA. Unfortunately, these effors are subject to the effectiveness of the compilers, i.e., the tuning process is addressing the question: "what kernel source manipulates the compiler into producing the best ISA?". The benefit of assembly is control over resource allocation and instruction scheduling, but at the sacrifice of being hardware specific.

Moreover, prior research focused primarily on how to achieve peak performance for a particular architecture using relatively large problem sizes which are amenable to achieving peak performance [15], or searching for a single GEMM kernel which is portable across architectures and problem sizes [16].

III. Problem Formulation

GEMM's genericity necessitates handling trillions of possible input combinations (precisions*transposeA*transposeB*M*N*K*lda*ldb*ldc) behind a single innocuous API. Due to GPU architecture, not all problem sizes are handled in the same manner; different problem sizes results in different: numbers of workgroups, assignments of workgroups to compute units (CU), memory addresses being requested concurrently, L1 and L2 cache lines being accessed by workgroups, effective global memory bandwidth due to cache reuse and ability to hide memory latencies by switching among concurrent workgroups on a CU. Therefore, achieving the highest degree of GEMM performance on a GPU must move past tuning a single kernel to fit the GPU architecture under ideal conditions, to curating a collection of size-tuned kernels, each of which is capable of maximizing performance for a family of problem sizes.

On top of maximizing performance, it is helpful to maximize functionality by reusing the tuned GEMM kernels for higher dimensional tensor contractions. For example, a GEMM kernel can be expanded to use a multi-dimensional summation rather than the traditional 1-dimensional summation along K. Also, multi-dimensional batching can expand the usability of current 1-dimensional batching.

IV. Methods

Tensile is a tool for auto tuning kernels for not only GEMM, but for higher dimensional tensor contractions too. Tensile's use is governed by a configuration file in YAML format. The configuration file specifies what to benchmark, how to analyze the benchmark data, and how to write the final results into a C library backend.

GEMMs and tensor contraction problems are described by two parts: the problem type and the problem sizes. The problem type consists of the precision and the tensor indices, i.e., which indices of tensors A and B get reduced together and to which indices of tensor C do they get written; for

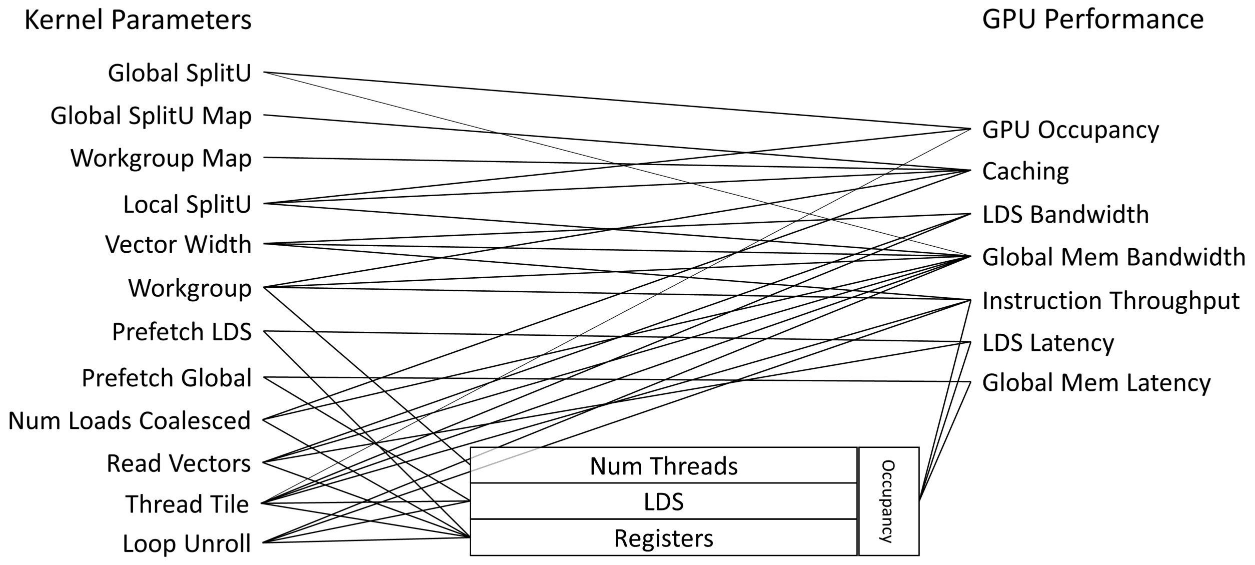


Fig. 1. Interdependencies of kernel parameter and GPU performance limitations; changing a single kernel parameter can improve and worsen competing performance limitations leading to a non-linear change in overall performance.

GEMM these indices describe the transposes. Specifically, Tensile describes a problem type using the parameters:

- *Precision:* Half, single, double and complex precisions are supported.
- *IndexAssignmentsA,B:* An array specifying how the indices of the A,B tensors map to either the C tensor indices or the summation indices.
- *NumIndicesC:* A scalar specifying how many indices the C tensor has.

For example, *Precision*=S, *IndexAssignmentsA*=[0, 3, 2], *IndexAssignmentsB*=[1, 3, 2], and *NumIndicesC*=3 represents $C_{ijk} = \sum_l A_{ilk} * B_{jlk}$, i.e., batched sgemm with matrix B transposed and the last dimension of each matrix being the batched dimension. As a more complicated example, *Precision*=H, *IndexAssignmentsA*=[6, 5, 0, 1, 4, 3], *IndexAssignmentsB*=[6, 5, 1, 4, 2, 3], and *NumIndicesC*=4 represents $C_{ijkl} = \sum_{mnl} A_{onijml} * B_{onjmkl}$, which is a 7-dimensional tensor contraction with a 3D summation and 1-dimensional batching; such a tensor contraction can be employed for batched image convolutions where the images and filters are 3-dimensional (row, col, channel). Dimensions are lableled in order of shortest stride to largest stride.

There are three types of indices which appear in these equations:

- *Batch* indices are recognized as indices which appear in tensors A, B and C and represent batching of multiple independent operation into a single kernel for higher gpu utilization.
- *Summation* indices are indices which appear in tensors A and B but not in C; these are the dimensions reduced and, therefore, do not appear in the C tensor.
- *Free* indices are a pair of indices, such as i and j, which both appear in the C tensor but only one appears in tensor A and one in tensor B.

Tensile maps multi-dimensional tensor contractions to fixed-dimensional GPU kernels through three special dimensions: dimension 0, dimension 1, and dimension U (for "unrolled"), hereafter written d0, d1 and dU. d0 and d1 are mapped to the two free indices with d0 being the dimension which has shorter stride in tensor C, and these are the two dimensions of the tiling of the C tensor. dU is taken to be the last summation index. All other summation indices are implemented as nested loops around the unrolled summation loop while all other batch indices are compacted into dimension 2 of the workgroup and grid of the gpu kernel.

For a given problem type, there are many problem sizes to be considered; a kernel for a problem type is valid for all problem sizes but won't necessarily be the fastest for all problem sizes. The problem sizes consists of a size for each of the problem indices and stride for each of a tensor indices. For the batched GEMM problem $C_{ijk} = \Sigma_l A_{ilk} * B_{jlk}$, the problem sizes are sizeI, sizeJ, sizeK, sizeL, while the strides are strideCI, strideCJ, strideCK, strideAI, strideAL, strideAK, strideBJ, strideBL and strideBK; typically strideCI, strideAI and strideBJ are 1 as is the case for GEMM. In order to be generalizable to high dimensional tensor contraction, Tensile doesn't use the traditional terminology of M, N, K, rows or columns. To simplify benchmarking, Tensile assumes that the data is compact such that strides are the natural product of the prior sizes.

The brute force methodology espoused by Tensile to create optimal GEMM kernels for all problem sizes and for any GPU, contains three parts:

1) Enumerate a user-defined set of kernels for a given problem type.

2) Benchmark a user-defined family of problem sizes, against the set of kernels.
3) Create a mapping of problem sizes to their optimal kernel.

A. GPU Performance Parameters

While GPUs are capable of massive economic performance, it comes at the cost of very simple cores and a high throughput, not low latency, oriented architecture. Well designed algorithms must be very architecture aware in order to achieve maximal efficiency. GPU architecture poses several potential bottlenecks which engineers must navigate:

- *Global Memory Bandwidth.* Data can be loaded from main memory to on-chip memory at a maximum rate of hundreds of gigabytes per second. To achieve this peak rate, memory operations must be as coalesced as possible, meaning adjacent threads operate on adjacent addresses.
- *Global Memory Latency.* The latency for accessing data from global memory is hundreds of cycles; this latency, as well as all memory latencies, can be hidden either through instruction level parallelism or a high occupancy of concurrently scheduled and interchangeable workgroups per CU.
- *Compute-Unit Occupancy.* Hiding the high memory latencies requires high occupancy, meaning a large number of workgroups that can be concurrently scheduled on a CU; the number of workgroups that can occupy on a CU is determined by the resources, such as LDS and vector general purpose registers (VGPR) on a CU divided by the resources used by each workgroup. Therefore, any kernel features which consume more resources can hurt occupancy.
- *Caches.* Since global memory can be too slow to support a fast GEMM algorithm, caches must be efficiency utilized to increase the effective bandwidth as well as lower the latency.
- *LDS Bandwidth.* The bandwidth of on-chip shared local memory (LDS) is several terabytes per second if bank conflicts are avoided.
- *LDS Latency.* The latency of on-chip memory is only many tens of cycles.
- *Instruction Divergence.* Divergent branches, where different threads within a workgroup perform different operations, slow throughput because the different instructions must be serialized.
- *Whole-GPU Occupancy.* To fully utilize a GPU, a kernel must contains enough workgroups to fill the entire GPU. As problem sizes remain steady but GPUs continually grow in CU count, this becomes increasingly strained.
- *Instruction Throughput.* To maximize performance, a kernel must maximize throughput of the multiply-accumulate (MAC) instructions which are actually counted for efficiency, i.e. 2*M*N*K; this can be done in two ways. First, minimize the number of non MAC instructions, such as memory operations, incrementing addresses, loop control logic or synchronization. Second, because GPUs are able to dual-issue instructions of different types (branches, scalar ALU or memory, vector ALU, vector memory, LDS, ...) from different wavefronts, the cost of the non-MAC instructions can be hidden if they can be scheduled simultaneously with MAC instructions. Achieving a high degree of dual-issuing requires well designed assembly code, of which the user is in control, and a well designed GPU architecture, of which the user is not in control.
- *Power.* GEMM tends to draw extremely high power since it runs many of the GPU's systems, LDS, caches, global memory and, especially, the VALUs, near peak rates. While GPUs typically have enough power to concurrently run many subsystems near peak, they may not have sufficient power to run all systems required by GEMM at full rate. Over-drawing power can necessitate the GPU to drop its clock speeds which directly reduces performance.

B. Kernel Parameters

Tensile employs several parameters to describe its kernels; each parameter both helps and hurts multiple GPU performance aspects as illustrated in Figure 1, thus necessitating automated tuning to search for the best combinations of kernel parameters. Tensile's main kernel parameters affecting performance are:

- *WorkGroup:* The size and shape of the workgroup assigned to share LDS; larger workgroups generally improve global data sharing, thus decreasing the demand on global memory bandwidth, but can lower instruction throughput as they require additional synchronization, and can lower occupancy due to requiring more LDS per workgroup. The number and shape of workgroups simultaneously scheduled on the GPU affects caching and instruction throughput.
- *LocalSplitU:* Splits up the summation within a workgroup; increasing this splitting will increase whole GPU occupancy, but at the cost of reducing global data sharing. A workgroup configuration of 8x8x4 denotes a *LocalSplitU*=4, i.e., the 256-threads are split up into four 8x8 subgroups, whose partial summations are reduced in LDS, before being written to global memory.
- *ThreadTile:* The size and shape of the C tensor's tiles assigned to each thread. Larger thread tiles increase global data sharing and local data sharing, increase instruction throughput but reduce the total number of workgroups. Additionally, larger thread tiles can reduce the CU occupancy as they require more registers and LDS.
- *MacroTile:* The product of the *ThreadTile* and *WorkGroup*, i.e., the whole tile of C elements computed by a single workgroup.
- *VectorWidth:* How the compute assignments and memory operation assignments are assigned to threads; widths of 2 and 4 denote operations on data types of float2 and float4, respectively. Larger vector widths improve the speed of both global and local memory operations as well increase

instruction throughput by reducing memory instruction count.

- *GlobalSplitU:* Splits up the summation among workgroups; increasing this splitting also increases whole GPU occupancy without reducing global data sharing, but the cost is having to perform the reduction via global memory atomics which increases demand on global memory bandwidth.
- *GlobalSplitUWorkGroupMap:* Workgroups assigned to the same tile are assigned round-robin or contiguously; this parameter affects L2 caching and contention of the global memory atomics.
- *PrefetchGlobalRead:* Summation loop will prefetch, i.e. double buffer, data from global memory to LDS one iteration in advance; doing so improves global memory latency hiding at a cost of consuming additional LDS.
- *PrefetchLocalRead:* Manually unrolled iterations within the summation loop will prefetch data from LDS into registers one iteration in advance; doing so improves LDS latency hiding but at a cost of consuming additional registers.
- *WorkGroupMapping:* Manipulates how workgroups are assigned to C matrix tiles which affects L2 caching.
- *LoopUnroll:* How many iterations to manually unroll the inner loop; higher unroll amounts can improve global memory bandwidth and instruction throughput but at a cost of requiring additional LDS and more registers for storing both extra global memory addresses and data loaded from global memory.
- *DepthU:* The product of *LoopUnroll* and *SplitU*, i.e., how many elements are processed in the unrolled dimension during each summation loop iteration.
- *NumLoadsCoalesced:* Governs how threads are assigned elements to load from global memory into local memory; it impacts global memory coalescing and L1 caching.
- *GlobalReadCoalesceGroup:* When data must be transposed, this affects whether data is read coalesced from global memory and is written scattered to LDS or read scattered from global memory and written coalesced to LDS, which affects both global memory and LDS bandwidth.
- *GlobalReadCoalesceVector:* When data must be transposed, this parameter determines whether threads will read whole vectors from global memory and write the vector components to LDS, or read the vector components from global memory and write the whole vectors to LDS. This affects global memory and LDS bandwidth as well as the number of registers.
- *KernelLanguage:* Language can be assembly or source (OpenCL or HIP). The benefit of generating assembly kernels is being able to control both register allocation and instruction scheduling. Both of these are of particular benefit to the highly efficient large *MacroTiles*, which require many VGPRs.

Figure 1 illustrates the main relationships between kernel parameters and GPU performance obstacles. All kernel parameters can be uniquely abbreviate by their capital letters, i.e., GSU stands for *GlobalSplitU*.

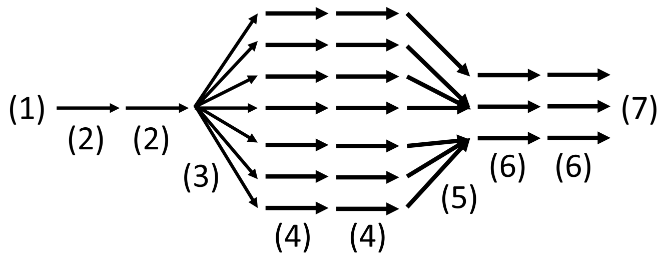


Fig. 2. Tensile's 7 step programmable benchmarking protocol first generates fast kernel candidates in Steps (1)-(6), then benchmarks those kernel candidates against a set of problem sizes in Step (7).

Kernel Parameters Not Explored: There are many more possible parameters to configure a kernel which Tensile doesn't use. For example, there are many more data flow patterns, i.e., how threads are assigned to transfer data from global memory to LDS, how threads are assigned to compute elements of the C matrix, and which threads are responsible for writing elements to the C matrix; no alternative memory management scheme showed enough competitive promise to implement and test.

Another possibility is to write a kernel which doesn't use LDS at all; such a kernel would rely on caching completely, rather than LDS, to achieve sufficiently high memory throughput. Early experiments showed that this alternative wasn't generally promising as LDS has much higher bandwidth than L2 cache, which is required for many problem sizes.

Also, it is possible to tile along the summation dimension, i.e., having the kernel read and write partial summation results to and from global memory. Doing so may improve L2 caching but would likely also increase overall demand on global memory bandwidth due to the additional traffic, thus worsening overall performance.

C. Benchmarking Protocol

To determine which kernel is optimal for a given problem size, Tensile employs a 7-step programmable benchmarking protocol which allows users to customize the process of searching for optimal kernels.

Step 1: Initial Solution Parameters: Before Tensile is able to benchmark a kernel parameter in Step 2 of Figure 2, such as *PrefetchGlobalRead*={False, True}, all other kernel parameters not being benchmarked must be specified. Therefore, the first step is to initialize a list of default kernel parameters, then subsequent steps of benchmarking will override a parameter from this default list, with the parameter determined from benchmarking. Tensile is pre-loaded with default parameters for any unspecified during tuning.

Step 2: Benchmark Common Parameters: Benchmarking common parameters determines parameters which are universally preferable to their alternatives regardless of other parameters. To benchmark common parameters:

(a) User specifies parameters and values to benchmark.
(b) Tensile benchmarks all parameter combinations for a user-specified problem size.
(c) Tensile selects the fastest parameter combination which is now labeled *determined* and will subsequently be used.

In practice, this parameters isn't used, since globally prefered parameters are set as defaults in Tensile and don't need to be re-benchmarked.

Step 3: Fork Parameters: Rather than continuing to determine globally fastest parameters, which eventually leads to a single fastest kernel, *forking* creates many different kernels, all of which will be considered for use. All *forked* parameters are considered *determined*, i.e., they arent benchmarked to determine which is fastest. Figure 2 shows 7 kernels being *forked* in Step 3.

Step 4: Benchmark Fork Parameters: Next, tuning continues its refinement by determining fastest parameters for each *forked* permutation, same as in Step 2.

Step 5: Join Parameters: After tuning the *forked* kernels, *joining* reduces the list of kernels so that fewer kernels will be considered for final use. Each kernel in the resulting list must have different values for the listed *JoinParameters*, for example, employing *JoinParameters* = *MacroTile* will result in only a few final kernels, each with a different *MacroTile*. If there are multiple kernels with the same *MacroTile*, only the fastest is kept. In Figure 2 the 7 *forked* kernel have been reduced to 3 *joined* kernels.

Step 6: Benchmark Join Parameters: Users can further tune parameters of the *joined* kernels. This steps is same as Steps 4 except that it tunes after joing so that there are fewer kernels to be tuned. In practice, this step isn't used; using Step 4 is preferred so that all parameters are benchmarked before *joinning*.

Step 7: Benchmark Final Parameters: At the conclusion of Step 6, all parameters of all kernels have been *determined* and the final set of kernels for consideration has been established. Now all final kernels will be benchmarked against all problem sizes specified by the user. Problem sizes can be specified as *Range* sizes and *Exact* sizes. *Range* sizes cause benchmarking of a broad range of sizes, and Tensile will be able to interpolate which kernel is best even between the specifically benchmarked sizes. *Exact* sizes cause a single problem size to be benchmarked, and the final library is guaranteed to choose the fastest kernel for that size. This final benchmarking generates the data that is subsequently analyzed for creating the mapping of problem size to optimal kernel.

Problem Size Groups: Problem size groups are designed to reduce the benchmarking computational complexity. User can curate multiple sets of kernels and benchmark them against multiple groups of problem sizes.

D. Library Logic

After the final benchmarking phase, Tensile has produced a database of kernels, problem sizes and the performance of every kernel for every problem size. To create the mapping of problem size to optimal kernel, Tensile employs a two-pronged approach for *exact* problem sizes and *range* problem sizes.

For *exact* problem sizes, Tensile simply selects which kernel was the fastest, and stores the one-to-one mapping of problem size to kernel and writes the library selection logic in the C langage:

```
if (sizeI==128 && sizeJ==128 && sizeK==128)
  return function_ptr_0;
```

Range problem sizes are more sophisticated since they must interpolate between benchmarked problem sizes to return the fastest kernels over a broad range. Tensile employs a comprehensive recursive size splitting algorithm which looks like,

```
if (sizeI < threshold_I_0) {
  if (sizeJ < threshold_J_0) {
    if (sizeK < threshold_K_0)
      return function_ptr_1;
    if (sizeK < threshold_K_1)
      return function_ptr_2;
    // more K thresholds
    return function_ptr_3;
  } // end J 0
  if (sizeJ < threshold_J_1) {
    // more K thresholds
  } // end J 1
  // more J, K thresholds
} // end I 0
if (sizeI < threshold_I_1) {
  // more J, K thresholds
} // end I 1
// more I, J, K thresholds
```

Simpler algorithm were tested to more elegantly represent GPU behavior, such as kernel selection is only a function of the quantity SizeI*SizeJ. However, the nonlinear behavior of GPUs necessitated the robust recursive strategy; this necessity will be illustrated later in Figure 4. The resulting library C code can be so long that it takes hundreds of microseconds during runtime for Tensile to lookup which kernel is fastest; for this reason, after a kernel is looked up for a problem size, the resulting mapping of that problem size to that kernel is cached in a std::map so that subsequent look-ups will only require a few microseconds. This small lookup penalty is far outweighed by the performance improvement of the problem size tuned kernels as will be shown later.

E. Experiments

To explore both Tensile's ability to generate high performance kernels and the necessity of generating multiple kernel parameter combinations for the many problem sizes, two experiments are conducted here. The first experiment generates a list of ˜100 kernels, and benchmarks them against a problem size range of M=[16, 5632], N=[16, 5632], K=[3104] and explores both the achieved performance for each problem size and which kernel is fastest for each problem size. For this experiment, the following kernel parameters were examined: *TT*={ 8x8, 8x4, 8x2, 4x8, 4x4, 4x2, 2x8, 2x4, 2x2 }; *WG*={ 16x16x1, 16x8x2, 8x16x2, 8x8x4 }; *GSU*={ 1, 2, 4, 6, 8 }; *DU*={ 8 }; *VW*={ min(4,*ThreadTile*[0],*ThreadTile*[2]) }; *PGR*={ True }; *PLR*={ True }; *WGM*={ 8 }.

In the second experiment, Tensile targets the DeepBench (https://github.com/baidu-research/DeepBench) list of GEMM problem sizes used in deep convolutional neural networks; Tensile begins with a list of thousands of kernels and selects from among them the fastest kernel for each of the ˜250 problem sizes. Performance of each size-tuned kernel is compared against performance of the single-tuned kernel. The single-tuned kernel is configured with *MT*=128x128, *DU*=8, *VW*=4 and *WGM*=8 and is known to achieve highest efficiency for large sgemm NT problems. For this experiment, the tunable Tensile kernel parameters are selected to be: *TT*={ 8x8, 8x6, 8x4, 8x2, 6x8, 6x6, 6x4, 6x3, 4x8, 4x6, 4x4, 4x3, 4x2, 3x6, 3x4, 3x3, 2x2 }; *WG*={ 32x4x2, 32x2x4, 24x8x1, 16x16x1, 16x8x2, 16x4x4, 16x2x8, 12x16x1, 8x16x2, 8x12x4, 8x8x4, 8x8x2, 8x8x1, 8x4x8, 6x32x1, 4x4x4 }; *GSU*={ 1, 2, 4, 6, 8 }; *DU*={ 8, 16, 32, 64 }; *VW*={ min(4,*TT*[0],*TT*[1]) }; *PGR*={ False, True }; *PLR*={ True }; *WGM*={ 1, 8 }. Tensile has a predetermined default value for all kernel parameters; all parameters not specified are assumed to be the default.

The GPU used for these experiments was an AMD Radeon Vega Frontier Edition air cooled (13.1 TFlops peak single precision throughput) on Ubuntu 16.04 running Radeon Open Compute Platform (ROCm) version 1.6.4 on an AMD Ryzen 7 1800X CPU. All performance measurements are based on kernel times read from device timers, i.e., no memory copies for kernel enqueueing latencies are included in the measurement, as to prevent CPU performance from affecting data. During tuning, Tensile sleeps the host process between benchmarks in order to give the GPU ample time to cool down and therefore keep the benchmark results consistent and order independent.

In order to discuss the results for different problem sizes, the following terminology will be used: MxN refers to specific individual sizes with values M and N or SizeI and SizeJ, while M*N refers to the collection of all MxN values whose product has the same value.

V. Results

Figure 3 shows the performance, in GFlops, achieved for each MxN problem size of Experiment 1. It can be seen that for small M*N, performance is quite low, well below 1 TFlops for many, while for M*N $>$ 700*700, performance has grown to over 7 TFlops, and for M*N $>$ 1936*1936 Tensile finally achieve 10 TFlops. Highest performance for this set of problem sizes was 11.7 TFlops for 3712x5632 which is ˜90% of the 13.1 TFlops peak performance of the device.

It is also significant that MxN=592x592, a square problem size, and MxN=64x5632, a skinny problem size having the same total size, achieve performances of 6.4 TFlops and 6.18 TFlops, respectively, demonstrating that performance is mainly determined by the product of M*N and not shape of the C matrix, provided M and N are sufficiently large as to not be too skinny, in this case 64. For much skinnier problem sizes, however, such high performance is no longer achievable; the skinniness precludes enough data reuse along the skinny dimension which leads to higher demand on global memory bandwidth.

Id	MT	DU	GSU	TT	VW	WG	LSU
0	64x32	8	4	4x2	2	16x16	1
1	32x32	8	6	2x2	2	16x16	1
2	32x128	8	8	2x8	2	16x16	1
3	32x32	8	8	2x2	2	16x16	1
4	32x64	8	8	2x4	2	16x16	1
5	64x64	8	2	4x4	4	16x16	1
6	64x128	8	1	4x8	4	16x16	1
7	32x64	8	6	2x4	2	16x16	1
8	32x32	8	8	2x4	2	16x8	2
9	64x32	8	4	4x4	4	16x8	2
10	32x128	8	6	2x8	2	16x16	1
11	32x32	8	6	2x4	2	16x8	2
12	64x32	8	8	4x2	2	16x16	1
13	32x64	8	4	2x8	2	16x8	2
14	32x64	8	4	2x4	2	16x16	1
15	32x32	8	2	2x4	2	16x8	2
16	128x64	8	1	8x4	4	16x16	1
17	128x32	8	6	8x2	2	16x16	1
18	32x32	8	4	2x2	2	16x16	1
19	64x32	8	2	4x2	2	16x16	1
20	128x128	8	2	8x8	4	16x16	1
21	128x32	8	8	8x2	2	16x16	1
22	128x32	8	4	8x2	2	16x16	1
23	64x32	8	6	4x2	2	16x16	1
24	64x64	8	1	4x4	4	16x16	1
25	64x64	8	6	4x4	4	16x16	1
26	32x32	8	4	2x4	2	16x8	2
27	128x128	8	1	8x8	4	16x16	1
28	64x64	8	4	4x4	4	16x16	1
29	32x64	8	2	2x4	2	16x16	1
30	32x128	8	4	2x8	2	16x16	1

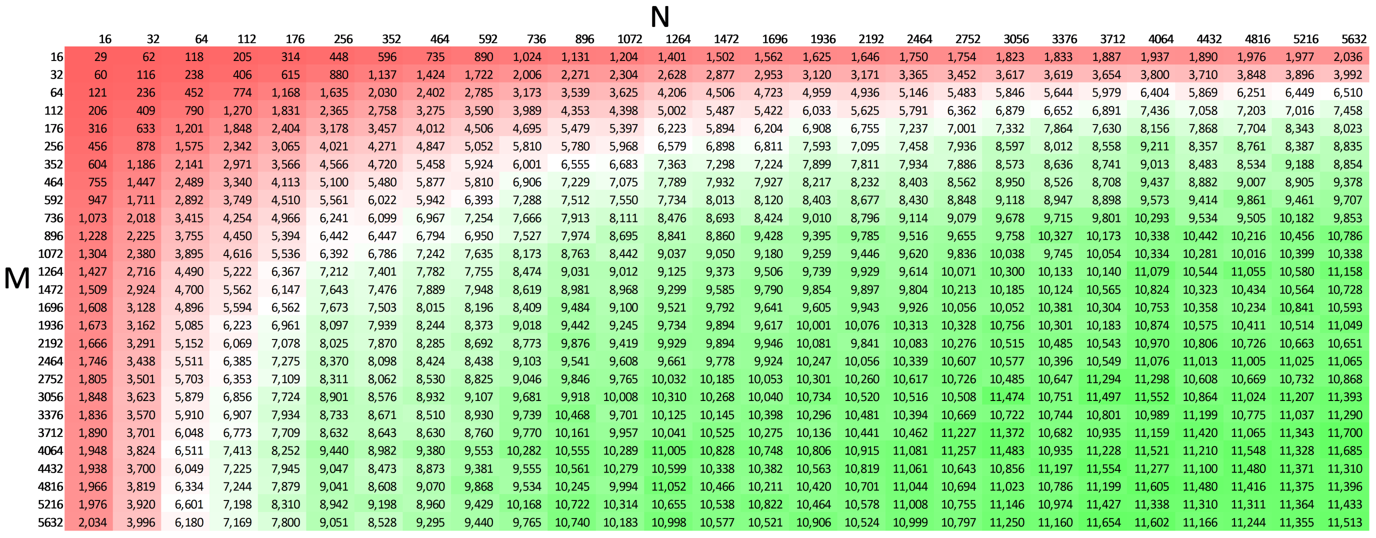
TABLE I
Experiment 1: Table of fastest kernel parameters used in Figure 4. Columns denote kernel id, *MacroTile*, *DepthU*, *GlobalSplitU*, *ThreadTile*, *VectorWidth*, *WorkGroup* and *LocalSplitU*.

Kernel 27 of Table I is the highest efficiency kernel; it uses a 128x128 *MacroTile* with an unroll depth of 8 and both global and local prefetching, same as the single-tuned reference kernel. It's large *MacroTile* and *ThreadTile* ensures global memory and LDS bandwidth won't bottleneck performance, while the global and local prefetching ensures that neither global memory latency nor LDS latency will be bottlenecks. Additionally, the *WorkGroupMapping*=8 used by all kernels encourages L2 cache tiling such that most data is read at L2 cache speeds which is typically twice the speed of global memory, further reducing the likelihood that memory bandwidth will hinder performance.

This 128x128 kernel's main loop contains

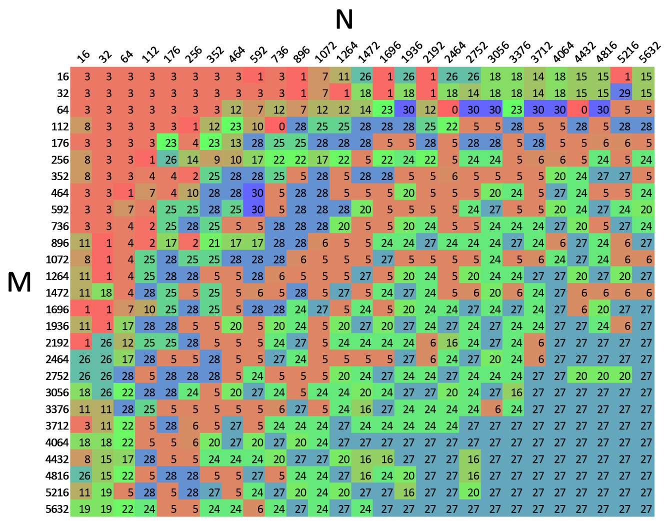
- 512 vector MAC instructions
- 8 other vector instructions
- 5 scalar loop control instructions
- 2 global memory instructions
- 37 LDS memory instructions

Because GPUs are able to dual-issue instructions, many of these operations' cost can be hidden, though not guaranteed. If all memory and scalar instructions were hidden, an efficiency of 512/(512+8)=98% would be expected. If only the memory instructions were hidden, an efficiency of 512/(512+8+5)=97% would be expected. If no instructions were hidden, an efficiency of 512/(512+8+5+2+37)=90% would be expected. Also contributing to achieved performance is sustained clock



	16	32	64	112	176	256	352	464	592	736	896	1072	1264	1472	1696	1936	2192	2464	2752	3056	3376	3712	4064	4432	4816	5216	5632
16	29	62	118	205	314	448	596	735	890	1,024	1,131	1,204	1,401	1,502	1,562	1,625	1,646	1,750	1,754	1,823	1,833	1,887	1,937	1,890	1,976	1,977	2,036
32	60	116	238	406	615	880	1,137	1,424	1,722	2,006	2,271	2,304	2,628	2,877	2,953	3,120	3,171	3,365	3,452	3,617	3,619	3,654	3,800	3,710	3,848	3,896	3,992
64	121	236	452	774	1,168	1,635	2,030	2,402	2,785	3,173	3,539	3,625	4,206	4,506	4,723	4,959	4,936	5,146	5,483	5,846	5,644	5,979	6,404	5,869	6,251	6,449	6,510
112	206	409	790	1,270	1,831	2,365	2,758	3,275	3,590	3,989	4,353	4,398	5,002	5,487	5,422	6,033	5,625	5,791	6,362	6,879	6,652	6,891	7,436	7,058	7,203	7,016	7,458
176	316	633	1,201	1,848	2,404	3,178	3,457	4,012	4,506	4,695	5,479	5,397	6,223	5,894	6,204	6,908	6,755	7,237	7,001	7,332	7,864	7,630	8,156	7,868	7,704	8,343	8,023
256	456	878	1,575	2,342	3,065	4,021	4,271	4,847	5,052	5,810	5,780	5,968	6,579	6,898	6,811	7,593	7,095	7,458	7,936	8,597	8,012	8,558	9,211	8,357	8,761	8,387	8,835
352	604	1,186	2,141	2,971	3,566	4,566	4,720	5,458	5,924	6,001	6,555	6,683	7,363	7,298	7,224	7,899	7,811	7,934	7,886	8,573	8,636	8,741	9,013	8,483	8,534	9,188	8,854
464	755	1,447	2,489	3,340	4,113	5,100	5,480	5,877	5,810	6,906	7,229	7,075	7,789	7,932	7,927	8,217	8,232	8,403	8,562	8,950	8,526	8,708	9,437	8,882	9,007	8,905	9,378
592	947	1,711	2,892	3,749	4,510	5,561	6,022	5,942	6,393	7,288	7,512	7,550	7,734	8,013	8,120	8,403	8,677	8,430	8,848	9,118	8,947	8,898	9,573	9,414	9,861	9,461	9,707
736	1,073	2,018	3,415	4,254	4,966	6,241	6,099	6,967	7,254	7,666	7,913	8,111	8,476	8,693	8,424	9,010	8,796	9,114	9,079	9,678	9,715	9,801	10,293	9,534	9,505	10,182	9,853
896	1,228	2,225	3,755	4,450	5,394	6,442	6,447	6,794	6,950	7,527	7,974	8,695	8,841	8,860	9,428	9,395	9,785	9,516	9,655	9,758	10,327	10,173	10,338	10,442	10,216	10,456	10,786
1072	1,304	2,380	3,895	4,616	5,536	6,392	6,786	7,242	7,635	8,173	8,763	8,442	9,037	9,050	9,180	9,259	9,446	9,620	9,836	10,038	9,745	10,054	10,334	10,281	10,016	10,399	10,338
1264	1,427	2,716	4,490	5,222	6,367	7,212	7,401	7,782	7,755	8,474	9,031	9,012	9,125	9,373	9,506	9,739	9,929	9,614	10,071	10,300	10,133	10,140	11,079	10,544	11,055	10,580	11,158
1472	1,509	2,924	4,700	5,562	6,147	7,643	7,476	7,889	7,948	8,619	8,981	8,968	9,299	9,585	9,790	9,854	9,897	9,804	10,213	10,185	10,124	10,565	10,824	10,323	10,434	10,564	10,728
1696	1,608	3,128	4,896	5,594	6,562	7,673	7,503	8,015	8,196	8,409	9,484	9,100	9,521	9,792	9,641	9,605	9,943	9,926	10,056	10,052	10,381	10,304	10,753	10,358	10,234	10,841	10,593
1936	1,673	3,162	5,085	6,223	6,961	8,097	7,939	8,244	8,373	9,018	9,442	9,245	9,734	9,894	9,617	10,001	10,076	10,313	10,328	10,756	10,301	10,183	10,874	10,575	10,411	10,514	11,049
2192	1,666	3,291	5,152	6,069	7,078	8,025	7,870	8,285	8,692	8,773	9,876	9,419	9,929	9,894	9,946	10,081	9,841	10,083	10,276	10,515	10,485	10,543	10,970	10,806	10,726	10,663	10,651
2464	1,746	3,438	5,511	6,385	7,275	8,370	8,098	8,424	8,438	9,103	9,541	9,608	9,661	9,778	9,924	10,247	10,056	10,339	10,607	10,577	10,396	10,549	11,076	11,013	11,005	11,025	11,065
2752	1,805	3,501	5,703	6,353	7,109	8,311	8,062	8,530	8,825	9,046	9,846	9,765	10,032	10,185	10,053	10,301	10,260	10,617	10,726	10,485	10,647	11,294	11,298	10,608	10,669	10,732	10,868
3056	1,848	3,623	5,879	6,856	7,724	8,901	8,576	8,932	9,107	9,681	9,918	10,008	10,310	10,268	10,040	10,734	10,520	10,516	10,508	11,474	10,751	11,497	11,552	10,864	11,024	11,207	11,393
3376	1,836	3,570	5,910	6,907	7,934	8,733	8,671	8,510	8,930	9,739	10,468	9,701	10,125	10,145	10,398	10,296	10,481	10,394	10,669	10,722	10,744	10,801	10,989	11,199	10,775	11,037	11,290
3712	1,890	3,701	6,048	6,773	7,709	8,632	8,643	8,630	8,760	9,770	10,161	9,957	10,041	10,525	10,275	10,136	10,441	10,462	11,227	11,372	10,682	10,935	11,159	11,420	11,065	11,343	11,700
4064	1,948	3,824	6,511	7,413	8,252	9,440	8,982	9,380	9,553	10,282	10,555	10,289	11,005	10,828	10,748	10,806	10,915	11,081	11,257	11,483	10,935	11,228	11,521	11,210	11,548	11,328	11,685
4432	1,938	3,700	6,049	7,225	7,945	9,047	8,473	8,873	9,381	9,555	10,561	10,279	10,599	10,338	10,382	10,563	10,819	11,061	10,643	10,856	11,197	11,554	11,277	11,100	11,480	11,371	11,310
4816	1,966	3,819	6,334	7,244	7,879	9,041	8,608	9,070	9,868	9,534	10,245	9,994	11,052	10,466	10,211	10,420	10,701	11,044	10,694	11,023	10,786	11,199	11,605	11,480	11,416	11,375	11,396
5216	1,976	3,920	6,601	7,198	8,310	8,942	9,198	8,960	9,429	10,168	10,722	10,314	10,655	10,538	10,822	10,464	10,578	11,008	10,755	11,146	10,974	11,427	11,338	11,310	11,311	11,364	11,433
5632	2,034	3,996	6,180	7,169	7,800	9,051	8,528	9,295	9,440	9,765	10,740	10,183	10,998	10,577	10,521	10,906	10,524	10,999	10,797	11,250	11,160	11,654	11,602	11,166	11,244	11,355	11,513

Fig. 3. Experiment 1: For each MxN problem size, the achieved performance, in GFlops, is charted. Figure 4 and Table I specify which kernel was used to achieve the specified performance.



	16	32	64	112	176	256	352	464	592	736	896	1072	1264	1472	1696	1936	2192	2464	2752	3056	3376	3712	4064	4432	4816	5216	5632
16	3	3	3	3	3	3	3	3	1	3	1	7	11	26	1	26	1	26	26	18	18	14	18	15	15	1	15
32	3	3	3	3	3	3	3	3	3	3	1	7	1	18	1	18	1	18	14	18	18	14	18	15	15	29	15
64	3	3	3	3	3	3	3	12	7	12	7	12	12	14	23	30	12	0	30	30	23	30	30	0	30	5	5
112	8	3	3	3	3	1	12	23	10	0	28	25	25	28	28	28	25	22	5	5	28	5	5	28	5	28	28
176	3	3	3	3	23	4	23	13	28	25	25	28	28	28	5	5	28	5	28	28	5	28	5	5	6	6	5
256	8	3	3	1	26	14	9	10	17	22	22	17	22	5	22	24	22	5	24	24	5	6	6	5	24	5	24
352	8	3	3	4	4	2	25	28	28	25	5	28	5	28	28	5	5	6	5	5	5	5	20	24	27	27	5
464	3	3	1	7	4	10	28	28	30	5	28	28	5	5	5	20	5	5	5	20	24	5	27	24	5	5	24
592	3	3	7	4	25	25	28	25	30	5	28	28	28	20	5	5	5	5	24	27	5	5	20	24	27	24	20
736	3	3	4	2	25	28	25	5	5	28	28	28	20	5	5	20	24	5	5	24	24	24	27	20	5	27	24
896	11	1	4	2	17	2	21	17	17	28	28	6	5	5	24	24	27	24	24	24	27	24	6	27	24	6	27
1072	8	1	4	25	28	25	25	28	28	28	6	5	5	5	27	5	5	5	5	24	5	6	24	27	24	6	6
1264	11	1	4	25	28	28	5	5	28	6	5	5	5	27	5	20	24	5	20	24	24	27	27	20	27	20	27
1472	11	18	4	28	25	5	25	5	6	5	28	5	27	5	24	27	24	5	6	20	6	24	27	6	6	6	6
1696	1	1	7	10	25	28	25	5	28	28	24	27	5	24	5	20	24	24	27	24	27	24	27	6	20	27	27
1936	11	1	17	28	28	5	5	20	5	20	24	5	20	27	20	27	24	27	24	27	24	24	27	27	24	6	27
2192	1	26	12	25	25	28	5	5	5	24	27	5	24	24	24	24	6	16	24	27	24	6	27	27	27	27	27
2464	26	26	17	28	5	5	28	5	5	27	24	5	5	5	5	27	6	24	27	20	24	6	27	27	27	27	27
2752	26	26	28	5	28	28	28	5	24	5	5	5	20	24	27	24	24	27	24	24	24	27	27	20	20	20	27
3056	18	26	22	28	28	24	5	20	27	24	5	24	24	20	24	27	27	20	24	27	16	27	27	27	27	27	27
3376	11	11	28	25	5	5	5	5	5	6	27	5	24	16	27	24	24	24	24	6	24	27	27	27	27	27	27
3712	3	11	22	5	28	6	5	27	5	24	24	24	27	24	24	24	24	24	27	27	27	27	27	27	27	27	27
4064	18	18	22	5	5	6	20	27	20	27	20	24	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
4432	8	15	17	28	5	5	24	24	24	20	27	27	20	16	16	27	27	27	16	27	27	27	27	27	27	27	27
4816	26	15	22	5	28	28	5	5	27	5	24	24	27	16	24	20	27	27	16	27	27	27	27	27	27	27	27
5216	11	19	5	28	5	28	27	5	24	27	20	24	20	27	27	16	27	27	20	27	27	27	27	27	27	27	27
5632	19	19	22	24	5	5	24	24	6	24	27	24	27	24	27	27	27	27	27	27	27	27	27	27	27	27	27

Fig. 4. Experiment 1: For each MxN problem size, the fastest kernel from Table I is charted. Some regions of problem sizes have the same fastest kernel (upper left and lower right) while other regions have different fastest kernels.

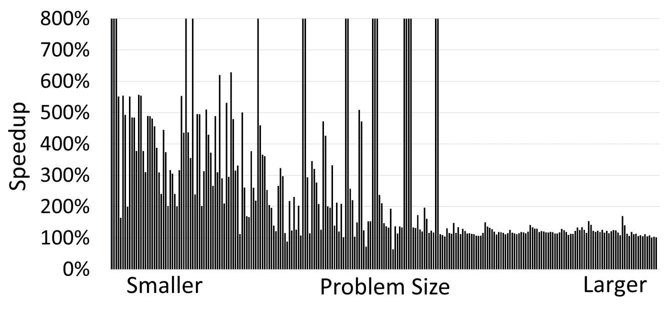


Fig. 5. Experiment 2: Speedup from using per problem size tuning versus using fastest *MacroTile*=128x128 kernel; 100% denotes no change in performance.

speed; the GPU has a boost clock speed of 1600MHz and a typical/base clock speed of 1382MHz. The estimated efficiency range of 90-98% combined with the GPU performance range of 13.1-11.3 TFlops suggests an optimal GEMM kernel should achieve between 10.2 and 12.8 TFlops. The highest performance reported in Figure 3 for the 128x128 kernel is 11.7 TFlops which is right in the middle of the expected range, suggesting that the kernel is indeed highly efficient, many but not all non-VALU operations are being hidden by dual-issuing and that the achieved clock speed is likely above average but below the boost clock.

Figure 4 illustrates which kernel from Table I is the fastest for each MxN problem size of Experiment 1. There exist many regions where the same kernel is fastest for the entire region, and there exist regions where the fastest kernel is different for the entire region. This is likely due to the nonlinear behavior of GPUs, for example, changing the tile size changes the number of workgroups resident on the GPU, which regions of the A and B tensors they read from and what the cache hit rate is. To capture the complexity of GPU performance which this map captures, Tensile has to employ the recursive threshold strategy outlined earlier, since kernel selection don't change monotonically with problem size.

Figure 5 portrays the speedup of the size-tuned kernels for the 250 DeepBench sizes of Experiment 2, over using a single-tuned kernel, with problem sizes being sorted from smallest to largest. The largest problems have speedups of 100-120%, with 100% representing equal performance to the single-tuned kernel, which is to be expected as the single-tuned kernel is tuned for large problem sizes. Many medium problem sizes have speedups of 150-200%, while small problem sizes have speedup of 200-500%. For only several kernels, speedups of 500-3500% were achieved.

Tables II and III lists the 144 of DeepBench's problems sizes which achieved the highest speedup from size-tuning;

the tables also list the parameters for each kernel, the achieved performance of the single-tuned kernel as well as the size-tuned kernel; the data is sorted in order of highest to lowest speedup. For the 250 different problem sizes benchmarked, roughly 150 unique size-tuned kernels were selected as fastest. The trend can be seen that large problem sizes, e.g. N N 2048x7000x2048 on the last row of Table III, can make use of large *MacroTiles* which have largest flops/byte ratios. The high ratio suggests the calculation should be compute bound and not memory bandwidth bound. Smaller problem sizes, such as N N 64x1x1216 on the 12th row of Table II, cannot make use of large *MacroTiles*, else they wouldn't produce enough workgroups to fill the entire GPU. The smaller tiles have worse flops/byte ratios and, therefore, those small problem sizes are more limited by global memory bandwidth.

Problem T N 35x8547x4096 near the middle of Table II demonstrates a speedup of 472% by employing a unusual *WorkGroup*=12x16 (composed of 3 64-thread wave-fronts) and a *ThreadTile*=3x3 resulting in a total *MacroTile*=36x48. That tile size fits the M=35 dimension closely so that very little computation is wasted, resulting in a much higher instruction throughput and efficiency than any traditional GEMM kernel centered around powers of 2 could achieve.

Problem N N 1024x8x500000 on row 13 of Table II exemplifies even more of Tensile's flexibility. The problem size is quite skinny and Tensile is able to create a *MacroTile*=128x8 which is also much skinnier than any traditional square GEMM tile, but matches the skinniness of the problem size to maximize efficiency. This problem size is also small, i.e., the tile by itself only generates 8 workgroups, which significantly underfills the GPU. Therefore, Tensile has employed a *LocalSplitU*=4 and *GlobalSplitU*=32 to split the large summation among many more workgroups; doing so enabled a 150X speedup.

VI. Conclusion

By generating, benchmarking and selecting highly tuned assembly kernels, Tensile is able to achieve very high performance for a broad range of problem sizes. In general, larger problem sizes can achieve higher performance through the use of larger *MacroTiles* which have lower demands on global memory bandwidth. The size-tuned kernels for small or skinny problem sizes can outperform the single-tuned kernel by 200-500% while kernels tuned for very small MxN but large K can improve performance up to 3500%, as is the case for the 150 size-tuned kernels targeting the DeepBench collection of problem size.

To precisely map problem sizes to optimal solutions, which mappings appear to be quite chaotic, Tensile has to employ a recursive size splitting algorithm rather than a more simplified model. The runtime cost of kernel lookup is negligible compared to the time savings of using optimal kernels. These technologies enable Tensile to significantly accelerate GEMM performance on modern GPUs for a variety of problem sizes.

Acknowledgment

Thank you to the other math library engineers Kent Knox, Andrew Chapman, Tinxing Dong and James Newling as well as to Greg Stoner, who have supported Tensile's development.

References

[1] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 25:1–25:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503219

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[4] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," *CoRR*, vol. abs/1709.03395, 2017. [Online]. Available: http://arxiv.org/abs/1709.03395

[5] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 4, pp. 511–515, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729

[6] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. E. Stone, and J. C. Phillips, "Quantifying the impact of gpus on performance and energy efficiency in hpc clusters," in *International Conference on Green Computing*, Aug 2010, pp. 317–324.

[7] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 673–686.

[8] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, Nov 2012.

[9] X. Cui, Y. Chen, C. Zhang, and H. Mei, "Auto-tuning dense matrix multiplication for gpgpu with cache," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, Dec 2010, pp. 237–242.

[10] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[11] A. Davidson and J. Owens, *Toward Techniques for Auto-tuning GPU Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–119. [Online]. Available: https://doi.org/10.1007/978-3-642-28145-7_11

[12] Y. Li, J. Dongarra, and S. Tomov, *A Note on Auto-tuning GEMM for GPUs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 884–892. [Online]. Available: https://doi.org/10.1007/978-3-642-01970-8_89

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[14] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2013, pp. 1–10.

[15] L. Jiang, C. Yang, Y. Ao, W. Yin, W. Ma, Q. Sun, F. Liu, R. Lin, and P. Zhang, "Towards highly efficient dgemm on the emerging sw26010 many-core processor," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 422–431.

[16] Y. M. Tsai, P. Luszczek, J. Kurzak, and J. Dongarra, "Performance-portable autotuning of opencl kernels for convolutional layers of deep neural networks," in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, Nov 2016, pp. 9–18.

TA	TB	M	N	K	MT	DU	GSU	TT	WG	LSU	WGM	F/B	Single	Size	Speedup
T	N	512	8	500000	64x8	64	16	4x4	16x2	8	1	4	32	1156	3662%
T	N	512	16	500000	64x16	64	16	4x4	16x4	4	1	6	62	2251	3633%
N	N	512	1	500000	64x8	32	16	4x4	16x2	8	1	4	5	158	3302%
N	N	512	2	500000	64x8	32	16	4x4	16x2	8	1	4	10	311	3249%
N	N	512	4	500000	64x16	64	16	4x4	16x4	4	1	6	19	618	3232%
N	N	512	8	500000	64x8	32	16	4x4	16x2	8	1	4	38	1199	3138%
N	N	512	16	500000	64x16	32	16	4x4	16x4	4	1	6	76	2329	3046%
T	N	1024	16	500000	64x16	64	32	4x4	16x4	4	1	6	123	2297	1860%
T	N	1024	8	500000	64x16	64	32	4x4	16x4	4	1	6	63	1167	1851%
N	N	1024	1	500000	128x8	32	16	4x4	32x2	4	1	4	10	152	1594%
N	N	1024	2	500000	128x8	32	16	4x4	32x2	4	1	4	19	303	1589%
N	N	1024	4	500000	128x8	32	16	4x4	32x2	4	1	4	38	604	1582%
N	N	1024	8	500000	128x8	32	16	4x4	32x2	4	1	4	76	1187	1553%
N	N	1024	16	500000	128x16	16	16	8x2	16x8	2	1	7	153	2325	1521%
N	N	128	1	1408	32x8	64	32	4x2	8x4	8	1	3	1	14	1385%
N	N	64	1	1216	32x8	32	16	2x2	16x4	4	1	3	1	7	1282%
N	N	128	1	1024	128x8	32	32	8x2	16x4	4	1	4	1	11	1136%
N	N	35	700	2560	32x32	32	4	4x4	8x8	4	8	8	180	1850	1028%
N	N	35	700	2048	64x16	16	4	4x4	16x4	4	8	6	170	1369	805%
N	N	35	1500	2560	36x48	16	8	3x3	12x16	1	1	10	376	2366	629%
N	N	35	1500	2048	64x16	16	4	4x4	16x4	4	1	6	324	2012	620%
N	N	512	16	512	16x16	16	4	2x2	8x8	4	8	4	56	314	557%
N	T	512	16	512	16x16	16	8	2x2	8x8	4	1	4	57	318	555%
N	N	512	2	512	32x8	32	16	2x2	16x4	4	1	3	7	40	554%
N	N	1760	16	1760	32x16	32	2	4x2	8x8	4	1	5	239	1325	553%
N	N	512	1	512	64x4	64	16	2x2	32x2	4	1	2	4	20	552%
N	N	512	4	512	32x8	32	8	2x2	16x4	4	1	3	14	79	552%
N	N	2048	32	2048	64x16	32	2	4x2	16x8	2	8	6	527	2802	531%
T	N	1760	32	1760	32x32	32	2	4x4	8x8	4	1	8	390	1990	510%
N	N	35	8457	4096	48x36	16	8	6x3	8x12	2	1	10	630	3204	509%
N	N	2560	32	2560	32x32	32	2	4x4	8x8	4	1	8	669	3350	501%
N	N	2048	16	2048	32x16	32	4	4x2	8x8	4	1	5	283	1402	496%
T	N	2048	16	2048	16x16	16	4	2x2	8x8	4	8	4	184	910	495%
N	N	1024	1	512	32x8	32	8	2x2	16x4	4	1	3	7	36	493%
N	T	512	32	512	16x16	16	4	2x2	8x8	4	1	4	113	554	489%
N	N	2560	16	2560	32x16	16	2	4x2	8x8	4	1	5	359	1753	489%
N	T	1024	16	512	16x16	32	4	2x2	8x8	4	1	4	118	579	488%
N	N	1024	2	512	32x8	32	2	2x2	16x4	4	1	3	15	72	485%
N	N	1024	4	512	32x16	16	4	4x2	8x8	4	1	5	29	143	484%
N	N	1024	16	512	16x16	32	8	2x2	8x8	4	1	4	117	563	481%
N	N	1760	64	1760	32x32	16	2	4x4	8x8	2	8	8	866	4152	479%
N	N	35	8457	2048	48x48	16	8	3x3	16x16	1	1	12	666	3147	472%
T	N	35	8457	4096	36x48	16	8	3x3	12x16	1	1	10	664	3135	472%
N	N	2048	64	2048	128x16	16	2	4x4	32x4	2	1	7	973	4470	460%
N	N	512	32	512	64x8	32	4	4x2	16x4	4	1	4	115	523	456%
N	T	1024	32	512	16x16	32	2	2x2	8x8	4	1	4	231	1029	445%
T	N	3072	16	1024	16x16	16	4	2x2	8x8	4	8	4	174	760	437%
T	N	1760	16	1760	32x16	16	8	4x2	8x8	2	1	5	200	872	436%
N	N	1760	32	1760	32x32	32	2	4x4	8x8	4	1	8	450	1934	429%
T	N	35	8457	2048	48x48	16	8	3x3	16x16	1	1	12	652	2778	426%
N	N	3072	4	1024	32x16	32	2	4x2	8x8	4	1	5	100	388	388%
N	N	3072	2	1024	32x16	32	2	4x2	8x8	4	8	5	50	189	378%
N	N	3072	1	1024	32x8	32	2	2x2	16x4	4	1	3	25	95	378%
N	N	128	1500	1280	64x32	16	2	4x4	16x8	2	1	11	1294	4876	377%
N	N	1024	32	512	32x16	16	2	4x2	8x8	4	1	5	233	869	374%
T	N	3072	32	1024	32x16	16	4	4x2	8x8	4	8	5	361	1343	373%
T	N	4096	16	4096	32x32	16	4	4x4	8x8	4	8	8	292	1068	366%
N	N	4096	16	4096	32x16	16	2	4x2	8x8	4	1	5	586	2118	361%
N	N	3072	16	1024	32x16	32	2	2x2	16x8	2	1	5	397	1412	355%
N	N	4096	32	4096	128x16	16	2	4x4	32x4	2	1	7	1092	3774	346%
N	N	35	8457	2560	48x36	16	4	6x3	8x12	2	1	10	1083	3600	332%
T	N	3072	64	1024	32x16	16	4	4x2	8x8	4	8	5	682	2260	331%
T	N	8448	16	2816	32x16	16	4	2x2	16x8	2	1	5	599	1936	323%
T	N	4096	32	4096	32x32	16	4	4x4	8x8	4	1	8	573	1837	321%
N	N	8448	1	2816	64x16	16	4	4x4	16x4	4	1	6	43	137	317%
N	N	8448	2	2816	64x16	16	4	4x4	16x4	4	1	6	86	273	317%
T	N	1760	64	1760	32x32	16	4	4x4	8x8	2	8	8	743	2339	315%
N	N	8448	4	2816	64x16	16	4	4x4	16x4	4	8	6	173	542	314%
N	N	4608	1	1536	32x8	32	16	2x2	16x4	4	1	3	39	121	310%
T	N	2560	16	2560	16x16	16	4	2x2	8x8	4	1	4	268	830	310%
N	N	4608	2	1536	32x8	32	16	2x2	16x4	4	1	3	78	241	309%
N	N	4608	4	1536	64x16	16	2	4x2	16x8	2	8	6	156	475	305%

TABLE II

EXPERIMENT 2: TABLE OF DEEPBENCH WINNING KERNELS USED IN FIGURE 5. COLUMNS REPRESENT TRANSPOSE A, TRANSPOSE B, M, N, K, *MacroTile*, *DepthU*, *GlobalSplitU*, *ThreadTile*, *WorkGroup*, *LocalSplitU*, *WorkGroupMapping*, FLOPS/BYTE RATIO, AND GFLOPS FOR SINGLE-TUNED KERNEL, SIZE-TUNED KERNELS THEN SPEEDUP (100% MEANS UNCHANGED).

TA	TB	M	N	K	MT	DU	GSU	TT	WG	LSU	WGM	F/B	Single	Size	Speedup
N	N	8448	16	2816	64x16	16	2	4x4	16x4	4	8	6	690	2052	297%
T	N	2048	32	2048	16x16	16	4	2x2	8x8	4	8	4	357	1056	295%
N	N	35	8457	1760	48x36	16	2	6x3	8x12	2	1	10	1355	3980	294%
N	N	4608	16	1536	32x16	16	2	4x2	8x8	2	1	5	620	1800	290%
N	N	2048	128	2048	32x32	16	2	4x4	8x8	2	8	8	1659	4596	277%
N	N	176	1500	1408	64x32	16	2	4x4	16x8	2	1	11	1840	4904	267%
N	N	3072	32	1024	64x16	16	2	4x2	16x8	2	8	6	744	1984	267%
T	N	2560	32	2560	32x16	16	4	4x2	8x8	4	8	5	529	1381	261%
T	N	6144	16	2560	32x32	16	4	4x4	8x8	4	1	8	532	1385	260%
N	N	4096	64	4096	64x32	32	2	4x4	16x8	2	8	11	1988	5115	257%
T	N	2048	64	2048	32x16	16	4	4x2	8x8	4	8	5	682	1730	254%
N	N	6144	1	2560	64x8	32	32	4x2	16x4	4	1	4	54	130	240%
N	N	6144	2	2560	32x16	16	2	2x2	16x8	2	1	5	108	259	240%
N	N	6144	4	2560	128x16	16	2	4x4	32x4	2	1	7	216	515	239%
T	N	4096	128	4096	32x32	16	4	4x4	8x8	2	8	8	1896	4503	238%
N	N	2560	64	2560	64x32	16	2	4x4	16x8	2	1	11	1237	2856	231%
T	N	4096	64	4096	32x32	16	4	4x4	8x8	4	8	8	1150	2543	221%
N	N	6144	16	2560	32x16	16	2	4x2	8x8	2	1	5	859	1883	219%
T	N	3072	128	1024	32x32	32	2	4x4	8x8	2	8	8	1192	2599	218%
T	N	8448	32	2816	32x32	32	2	4x4	8x8	4	1	8	1194	2540	213%
N	N	4096	128	4096	64x64	16	1	4x4	16x16	1	8	16	3176	6689	211%
T	N	4608	16	1536	16x16	16	4	2x2	8x8	4	1	4	412	866	210%
N	N	2560	128	2560	64x32	16	2	4x4	16x8	2	8	11	2134	4455	209%
T	N	2048	128	2048	32x32	16	2	4x4	8x8	4	8	8	1231	2567	209%
T	N	7680	16	2560	32x32	16	4	4x4	8x8	4	1	8	596	1223	205%
T	N	6144	32	2560	32x32	32	2	4x4	8x8	4	1	8	1095	2227	203%
N	N	7680	4	2560	128x16	16	2	4x4	32x4	2	1	7	269	543	202%
N	N	7680	1	2560	128x16	16	2	4x4	32x4	2	1	7	67	136	202%
N	N	7680	2	2560	128x16	16	2	4x4	32x4	2	1	7	135	271	201%
T	N	7680	32	2560	32x32	32	2	4x4	8x8	4	1	8	1215	2442	201%
N	N	4224	1	128	32x16	16	2	4x2	8x8	4	1	5	18	36	200%
N	N	7680	32	2560	64x32	32	2	4x4	16x8	2	8	11	1940	3816	197%
N	N	512	6000	2048	128x64	16	1	8x4	16x16	1	8	21	3901	7667	197%
N	N	7680	16	2560	128x16	16	2	4x4	32x4	2	1	7	1072	2101	196%
N	N	7680	128	2560	128x64	16	1	8x4	16x16	1	1	21	3100	5997	193%
N	N	3072	1500	1024	128x96	16	1	8x6	16x16	1	1	27	4876	8455	173%
N	N	5124	9124	4096	128x128	24	1	8x8	16x16	1	1	32	5731	9731	170%
N	N	4608	32	1536	64x32	16	2	8x4	8x8	2	8	11	1159	1962	169%
T	N	4608	32	1536	32x16	16	4	4x2	8x8	4	8	5	808	1347	167%
N	N	3072	1	128	16x16	8	2	2x2	8x8	2	1	4	13	21	165%
N	N	1024	3000	2048	128x64	16	1	8x4	16x16	1	1	21	5527	8929	162%
N	N	4096	7000	4096	128x128	24	1	8x8	16x16	1	1	32	6566	10111	154%
N	N	512	1500	2560	64x64	16	1	4x4	16x16	1	1	16	5242	8049	154%
N	N	512	1500	2048	128x64	16	1	8x4	16x16	1	1	21	4038	6189	153%
N	N	512	24000	2048	128x64	16	1	8x4	16x16	1	1	21	6276	9435	150%
N	N	512	1500	1536	64x64	16	1	4x4	16x16	1	8	16	4968	7443	150%
N	N	1024	6000	2048	128x96	16	1	8x6	16x16	1	1	27	6379	9443	148%
N	N	512	1500	2816	64x64	16	1	4x4	16x16	1	8	16	5577	8214	147%
T	N	4096	7000	4096	64x128	16	1	4x8	16x16	1	1	21	4385	6243	142%
N	N	512	48000	2048	128x64	16	1	8x4	16x16	1	1	21	6913	9770	141%
T	N	5124	9124	4096	64x128	16	1	4x8	16x16	1	1	21	4474	6276	140%
T	N	35	8457	2560	36x48	16	8	3x3	12x16	1	1	10	1084	1512	139%
T	N	1024	700	512	64x128	16	1	4x8	16x16	1	8	21	2754	3833	139%
N	N	1024	1500	2048	128x96	16	1	8x6	16x16	1	1	27	6045	8296	137%
N	N	1024	1500	1536	128x96	16	1	8x6	16x16	1	8	27	6308	8656	137%
N	N	1024	1500	2560	128x96	16	1	8x6	16x16	1	8	27	6608	9027	137%
T	N	512	24000	2048	64x64	16	1	4x4	16x16	1	1	16	5095	6960	137%
N	N	5124	1500	2048	128x64	16	1	8x4	16x16	1	8	21	6574	8862	135%
N	N	1024	48000	2048	128x128	16	1	8x8	16x16	1	1	32	7625	10260	135%
N	N	1024	24000	2048	128x96	16	1	8x6	16x16	1	1	27	7505	10059	134%
N	N	512	3000	2560	128x96	16	1	8x6	16x16	1	8	27	6557	8771	134%
N	N	512	3000	2816	128x96	16	1	8x6	16x16	1	8	27	6814	9104	134%
N	N	5124	9124	2048	128x128	24	1	8x8	16x16	1	1	32	7317	9772	134%
N	N	512	3000	1536	128x96	16	1	8x6	16x16	1	1	27	6205	8267	133%
T	N	2048	7000	2048	64x128	16	1	4x8	16x16	1	1	21	5204	6895	132%
N	N	1024	1500	2816	128x96	16	1	8x6	16x16	1	8	27	6978	9194	132%
N	N	3072	3000	1024	128x96	16	1	8x6	16x16	1	1	27	6787	8870	131%
T	N	1024	24000	2048	64x128	16	1	4x8	16x16	1	1	21	5698	7394	130%
T	N	512	48000	2048	64x64	16	1	4x4	16x16	1	1	16	5770	7477	130%
T	N	512	24000	1536	64x64	16	1	4x4	16x16	1	1	16	6335	8180	129%
T	N	3072	24000	1024	96x128	16	1	6x8	16x16	1	1	27	6363	8213	129%
N	N	2048	7000	2048	128x128	24	1	8x8	16x16	1	1	32	7582	9725	128%

TABLE III

EXPERIMENT 2: TABLE OF DEEPBENCH WINNING KERNELS USED IN FIGURE 5. COLUMNS REPRESENT TRANSPOSE A, TRANSPOSE B, M, N, K, *MacroTile*, *DepthU*, *GlobalSplitU*, *ThreadTile*, *WorkGroup*, *LocalSplitU*, *WorkGroupMapping*, FLOPS/BYTE RATIO, AND GFLOPS FOR SINGLE-TUNED KERNEL, SIZE-TUNED KERNEL THEN SPEEDUP (100% MEANS UNCHANGED).