Title
A High-Throughput Solver for Marginalized Graph Kernels on GPU

Authors
Tang, Yu-Hang
Selvitopi, Oguz
Popovici, Doru Thom
et al.

Publication Date
2020-05-22

DOI
10.1109/ipdps47924.2020.00080

Peer reviewed

A High-Throughput Solver for Marginalized Graph Kernels on GPU

Yu-Hang Tang, Oguz Selvitopi, Doru Popovici, Aydın Buluç
Computational Research Division, Lawrence Berkeley National Laboratory
Email: {tang, roselvitopi, dtpopovici, abuluc}@lbl.gov

***Abstract*—We present the design and optimization of a solver for efficient and high-throughput computation of the marginalized graph kernel on General Purpose GPUs. The graph kernel is computed using the conjugate gradient method to solve a generalized Laplacian of the tensor product between a pair of graphs. To cope with the large gap between the instruction throughput and the memory bandwidth of the GPUs, our solver forms the graph tensor product on-the-fly without storing it in memory. This is achieved by using threads in a warp cooperatively to stream the adjacency and edge label matrices of individual graphs by small square matrix blocks called tiles, which are then staged in registers and the shared memory for later reuse. Warps across a thread block can further share tiles via the shared memory to increase data reuse. We exploit the sparsity of the graphs hierarchically by storing only non-empty tiles using a coordinate format and nonzero elements within each tile using bitmaps. We propose a new partition-based reordering algorithm for aggregating nonzero elements of the graphs into fewer but denser tiles to further exploit sparsity.**

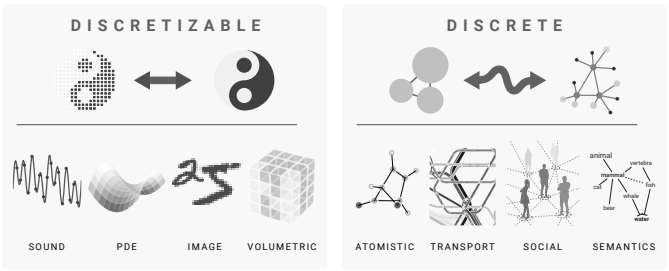
We carry out extensive theoretical analyses on the graph tensor product primitives for tiles of various density and evaluate their performance on synthetic and real-world datasets. Our solver delivers three to four orders of magnitude speedup over existing CPU-based solvers such as GraKeL and GraphKernels. The capability of the solver enables kernel-based learning tasks at unprecedented scales.

I. Introduction

Recent advances in machine learning have sparked unique opportunities for building artificial intelligence using graph data, which are powerful structures for representing non-sequential data of discrete nature. As illustrated by Figure 1, a distinction of graph-based discrete data from vector-based discretizable data is that the former consists of indivisible elements that must be inserted or withdrawn atomically, while the latter consist of discretized samples drawn from a continuous signal at tunable resolutions. Consequently, graph data does not trivially permit interpolation, convolution, and inner product, which are the operations commonly used in feature extraction. As a result, special care must be taken to generalize machine learning algorithms that operate on fixed-length feature vectors and uniform grids to their graph-based counterparts.

One way to interface graph data to machine learning algorithms is to apply the *kernel trick*. A *graph kernel* in this context refers to a function that performs inner product operations between graphs after implicitly transforming them into high- and even infinite-dimensional feature vectors. A valid graph kernel must be positive definite, meaning that the feature space must be a reproducing kernel Hilbert space.

Fig. 1. Image and voice recordings are *discretizable* objects in the sense that numeric representations for them can be obtained by sampling at various resolutions. Molecules and social networks are intrinsically non-sequential and discrete objects, and thus can be better represented by graphs.

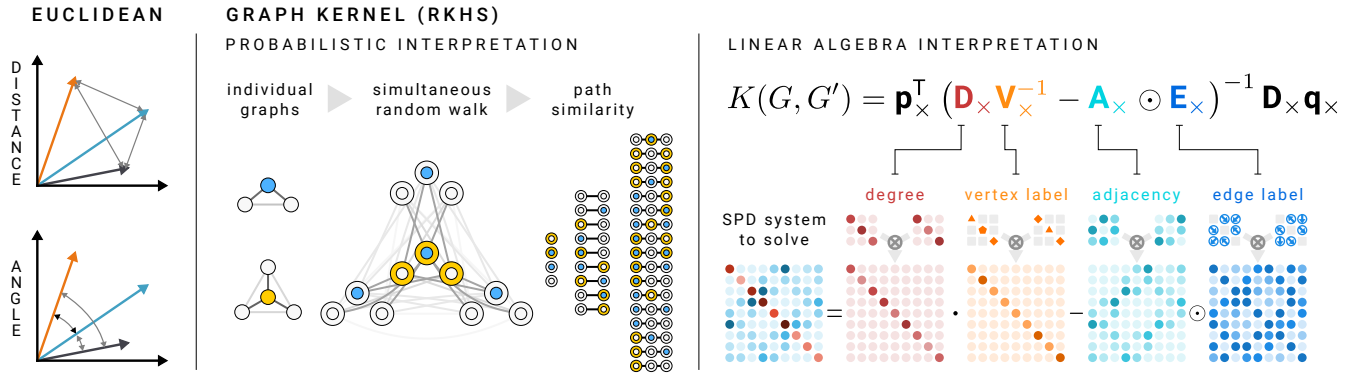


The inner product thus naturally induces a measure of graph similarity using the cosine of angles in the feature space. Graph kernels allow a wide range of kernel-based learning methods, *e.g.* support vector machine, Gaussian process regression, spectral clustering, principal component analysis, to operate straightforwardly on graph-based datasets.

The marginalized graph kernel [1] is a powerful tool for graph similarity comparison between labeled and weighted graphs of arbitrary size and topology. As illustrated in Figure 2, the kernel constructs a feature space containing infinitely many dimensions, each of which represents a path on a graph. The weight of a feature is set equal to the probability of its path in a Markovian random walk process induced by the graph's adjacency matrix. The overall similarity is then defined as the expectation of partial similarities between all pairs of same-length paths, each of which is computed as the product of a sequence of node-by-node and edge-by-edge comparisons. Besides computing an overall similarity score between two graphs, the kernel also defines a measure of node-wise similarity, which is the expectation of the similarities between all pairs of paths originating from a given pair of nodes. The node-wise similarity is particularly useful for learning tasks involving the transfer of node labels. The kernel has found successful application in tasks such as prediction of molecular energy [2] and protein function [3].

This paper focuses on the efficient and high-throughput computation of the marginalized graph kernel, which is necessary because the pairwise similarity matrix between *all pairs* of graphs has to be computed *repeatedly* with different hyperparameters for training many kernel-based methods. As will be shown later in Section II-B and Appendix A , each marginalized graph kernel evaluation between a pair of graphs

Fig. 2. Visualizing the Euclidean Kernels and the Graph Kernel.



involves solving a linear system whose size is the product of the number of nodes of the two graphs.

To obtain a pairwise similarity matrix for a dataset of 2000 graphs each with 100 nodes, we need to solve a million $10^4 \times 10^4$ linear systems. Clearly, a high-performance and high-throughput solver is crucial for applying and scaling the marginalized graph kernel to even larger datasets.

In this paper, we present a series of algorithms and optimizations, such as on-the-fly Kronecker product matrix-vector multiplication, graph reordering, and sparsity exploitation, to accelerate the marginalized graph kernel computation. The synergy of the algorithms leads to a solver that achieves a significant performance boost over existing packages on general-purpose graphics processing units (GPGPUs).

The rest of the paper is organized as follows. In Section II, we briefly review related mathematical background knowledge, introduce the formulation of the marginalized graph kernel, and carry out a preliminary analysis to identify design challenges. In Section III, we explore several design options of a dense Kronecker product matrix-vector multiplication primitive and identify the optimal one through Roofline analyses and microbenchmarking. In Section IV we examine data structure designs, graph reordering algorithms, and sparse Kronecker product matrix-vector multiplication primitives in order to exploit the sparsity in the graph. In Section V we present data sharing and load balancing approaches for scaling the algorithm onto entire GPUs. Benchmark datasets and results are given in section VI and section VII, respectively.

We discuss the connections between our project and previous ones in section VIII and conclude the paper in section IX.

II. Theoretical Background

A. Preliminaries and Notations

We use lower case letters in bold font, *e.g.* $\mathbf{a}$, to denote vectors, and upper case letters in bold font, *e.g.* $\mathbf{A}$, to denote matrices. By default, we assume vectors are column vectors. We use $\mathbf{diag}(\mathbf{a})$ to denote a diagonal matrix whose diagonal elements are specified by $\mathbf{a}$. We use *vertex* and *node* interchangeably to refer to the fundamental units of graphs.

Undirected graph: An undirected graph G is a discrete structure consisting of a set of uniquely-indexed vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of undirected edges $E \subset V \times V$. The vertices and edges may be labeled using elements from label sets Σ_v and Σ_e, respectively.

Weighted graph: In a weighted graph, each edge (v_i, v_j) is associated with a non-negative weight w_{ij}. In undirected graphs $w_{ij} = w_{ji}$. $w_{ij} = 0$ if v_i and v_j are not connected by an edge. An unweighted graph can be regarded as a specialized weighted graph where $w_{ij} = 1$ between each pair of (v_i, v_j) connected by an edge and 0 elsewhere.

Walk on graph: Two vertices are neighbors if they are connected by an edge. A walk on a graph is a sequence of vertices and edges such that all consecutive pairs of vertices are neighbors.

Adjacency matrix: The adjacency matrix of a graph of n vertices is a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ with $\mathbf{A}_{ij} = w_{ij}$. The adjacency matrices of undirected graphs are symmetric since $w_{ij} \equiv w_{ji}$.

Edge label matrix: The edge label matrix of a graph of n vertices is a matrix $\mathbf{E} \in {\Sigma_\mathrm{e}}^{n \times n}$ with $\mathbf{E}_{ij} = e_{ij}$. $\mathbf{E}$ has the same symmetry and sparsity pattern with $\mathbf{A}$.

Kronecker product: Given matrices $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{B} \in \mathbb{R}^{n' \times m'}$, the Kronecker product $\mathbf{P} = \mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{nn' \times mm'}$ is defined as:

$$\mathbf{P} = \mathbf{A} \otimes \mathbf{B} \coloneqq \begin{bmatrix} \mathbf{A}_{1,1}\mathbf{B} & \mathbf{A}_{1,2}\mathbf{B} & \ldots & \mathbf{A}_{1,m}\mathbf{B} \\ \mathbf{A}_{2,1}\mathbf{B} & \mathbf{A}_{2,2}\mathbf{B} & \ldots & \mathbf{A}_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{n,1}\mathbf{B} & \mathbf{A}_{n,2}\mathbf{B} & \ldots & \mathbf{A}_{n,m}\mathbf{B} \end{bmatrix}$$

To better visualize the correspondence between an element of the Kronecker product matrix and its source elements from the operand matrices, we use a quadruple index notation $\mathbf{P}_{ii',jj'}$, which is located at the $(i \times n + i')$-th row and $(j \times m + j')$-th column of $\mathbf{P}$, to denote the element formed by $\mathbf{A}_{ij} \cdot \mathbf{B}_{i'j'}$. Similarly, for a vector $\mathbf{p} = \mathbf{a} \otimes \mathbf{b}$, we use $\mathbf{p}_{ii'}$ to denote its $(i \times n + i')$-th component which is formed by $\mathbf{a}_i \cdot \mathbf{b}_{i'}$.

Generalized Kronecker product: Given a set $\mathbb{S}$ whose elements are necessarily numeric, a generalized Kronecker product $\mathbf{P} = \mathbf{A} \overset{\kappa}{\otimes} \mathbf{B}$ between two matrices $\mathbf{A} \in \mathbb{S}^{n \times m}$ and $\mathbf{B} \in \mathbb{S}^{n' \times m'}$ with respect to a kernel $\kappa : \mathbb{S} \times \mathbb{S} \to \mathbb{R}^+$, is a real matrix $\mathbf{P} \in \mathbb{R}^{nn' \times mm'}$ where $\mathbf{P}_{ii',jj'} = \kappa(\mathbf{A}_{ij}, \mathbf{B}_{i'j'})$. In other words, κ is a generalization of the real number multiplication operation as used in the standard Kronecker product on $\mathbb{S}$.

Hadamard (element-wise) product: The element-wise product, also known as the Hadamard product, between two matrices of the same size $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ is another matrix $\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{m \times n}$ with $(\mathbf{A} \odot \mathbf{B})_{ij} \coloneqq \mathbf{A}_{ij}\, \mathbf{B}_{ij}$.

B. Marginalized Graph Kernel

We have previously shown [2] that the computation to apply marginalized graph kernel between two labeled graphs G and G' can be simplified into solving a linear system involving a generalized Laplacian of the tensor product graph $G \otimes G'$:

$$K_{\mathrm{MG}}(G, G') = \mathbf{p}_\times^\mathsf{T} \left(\mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times\right)^{-1} \mathbf{D}_\times \mathbf{q}_\times. \quad (1)$$

Here

- $\mathbf{p}_\times \coloneqq \mathbf{p} \otimes \mathbf{p}'$ is the starting probability of a Markovian random walk process from each node of the product graph;
- $\mathbf{q}_\times \coloneqq \mathbf{q} \otimes \mathbf{q}'$ is the stopping probability of the random walk process on each node of the product graph;
- $\mathbf{A}_\times \coloneqq \mathbf{A} \otimes \mathbf{A}'$ is the adjacency matrix of the product graph;
- $\mathbf{D}_\times \coloneqq \mathbf{diag}(\mathbf{d} \otimes \mathbf{d}')$ is the degree matrix of the product graph, while $\mathbf{d}_i = \sum_j \mathbf{A}_{ij} + \mathbf{q}_i$ is the degree of node i;
- $\mathbf{V}_\times \coloneqq \mathbf{diag}\left(\mathbf{v} \overset{\kappa}{\otimes} \mathbf{v}'\right)$ is a diagonal matrix set by the generalized Kronecker product with respect to a vertex base kernel $\kappa_\mathrm{v} : \Sigma_\mathrm{v} \times \Sigma_\mathrm{v} \to \mathbb{R}^+$, while $\mathbf{v}$ and $\mathbf{v}'$ contains the vertex labels of G and G', respectively;
- $\mathbf{E}_\times \coloneqq \mathbf{E} \overset{\kappa}{\otimes} \mathbf{E}'$ is the generalized Kronecker product between edge label matrices E and E' with respect to an edge base kernel $\kappa_\mathrm{e} : \Sigma_\mathrm{e} \times \Sigma_\mathrm{e} \to \mathbb{R}^+$.

Our extended preprint [4] gives a detailed derivation of Equation (15). Our earlier work [2] contains an example of the rules for determining the specific values for $\mathbf{p}$, $\mathbf{q}$, $\mathbf{V}$, $\mathbf{A}$, and $\mathbf{E}$.

$\mathbf{D}_\times$ and $\mathbf{V}_\times$ are complete diagonal matrices, and $\mathbf{A}_\times$ and $\mathbf{E}_\times$ only have non-zero off-diagonal elements. The linear system in Equation (15) is symmetric and positive definite as long as the base kernels $\kappa_\mathrm{v}(\cdot,\cdot)$ and $\kappa_\mathrm{e}(\cdot,\cdot)$ themselves are positive definite with ranges within $(0, 1]$ and $[0, 1]$, respectively. The actual arithmetics involved to compute κ_v and κ_e determine the computational costs to generate $\mathbf{E}_\times$ and $\mathbf{V}_\times$, an important factor that will be considered in Section III for designing efficient matrix-vector multiplication primitives.

A degenerate case worth noting is when both the graph nodes and edges are unlabeled. This eliminates the base kernels as well as the $\mathbf{V}_\times$ and $\mathbf{E}_\times$ matrices from Equation (15). Consequently, Equation (15) gets simplified into

$$K_{\mathrm{RW}}(G, G') = \mathbf{p}_\times^\mathsf{T} \left(\mathbf{D}_\times - \mathbf{A}_\times\right)^{-1} \mathbf{D}_\times \mathbf{q}_\times. \quad (2)$$

Equation (2) is essentially the random walk graph kernel proposed in [5]. We denote it as the *unlabeled* graph kernel, and use it as one of the two model problems in performance modeling and algorithm design.

C. Preconditioned Conjugate Gradient Method

The linear system in Equation (15) can be solved by a variety of methods such as conjugate gradient (CG), spectral decomposition, fixed-point iteration, and generalized Sylvester equation [5], [6]. Among those, spectral decomposition delivers the best performance *if* the edges are unlabeled or labeled with a small set of distinct elements. CG is often favored in many real-world applications where the edges are labeled using larger and more complex attribute sets. For example, the edges can be labeled by interatomic distances that span some continuous interval of $\mathbb{R}^+$ when the graphs are used to represent 3D structures of molecules [2]. In this case, the spectral decomposition method is no longer advantageous due to the need for looping over all pairs of distinct labels.

Algorithm 1 illustrates the application of CG, together with a diagonal preconditioner, for solving Equation (15). Being formulated as an exact solver for symmetric and positive definite linear systems that iteratively minimizes a residual vector in successive orthogonal directions, the method has in practice often being used as an iterative solver because the convergence can be achieved quickly due to the orthogonalization of successive search directions.

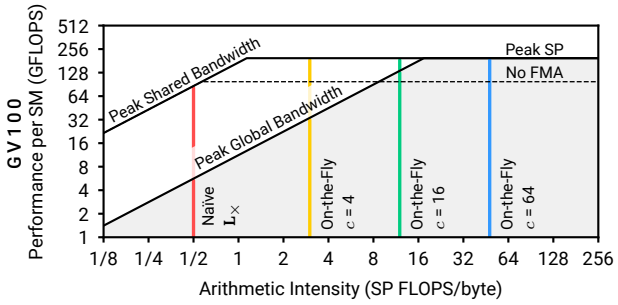
Algorithm 1 Preconditioned conjugate gradient algorithm for the marginalized graph kernel. Legend: ▩·▮: off-diagonal symmetric matrix-vector multiplication, ▨·▮: diagonal matrix-vector multiplication, ▮ᵀ·▮: vector dot product, ▮+▮: (scaled) vector addition.

```
1  function CG4GK(d,d′,v,v′,A,A′,E,E′, q,q′)
2      M ← diag[ ( d ⊗ d′) ⊙ ( v ⊗κ v′)^−1 ]                    ▮+▮
3      x ← 0                                                    ▮+▮
4      r ← (d ⊗ d′) · (q ⊗ q′)                                  ▨·▮
5      z ← v ⊗κ v′                                              ▮+▮
6      p ← z                                                    ▮+▮
7      ρ ← rᵀz                                                  ▮ᵀ·▮
8      repeat
9          a ← ( d ⊗ d′) ⊙ ( v ⊗κ v′)^−1 · p                    ▨·▮
10             +(A ⊗ A′) ⊙ (E ⊗κ E′) · p                        ▩·▮
11         α ← ρ/(pᵀa)                                          ▮ᵀ·▮
12         x ← x + αp                                           ▮+▮
13         r ← r − αa                                           ▮+▮
14         z ← M^−1 r                                           ▮+▮
15         ρ′ ← rᵀz                                             ▮ᵀ·▮
16         β ← ρ′/ρ
17         p ← z + βp                                           ▮+▮
18         ρ ← ρ′
19     until rᵀr < ϵ
20     return x
```

D. Preliminary Roofline Analysis

The Kronecker product matrix-vector multiplication operation $(\mathbf{A} \otimes \mathbf{A}') \odot (\mathbf{E} \overset{\kappa}{\otimes} \mathbf{E}') \cdot \mathbf{p}$, as highlighted on line 10 of Algorithm 1, has the highest order of asymptotic complexity of $\mathcal{O}(N^2)$ for a $N \times N$ system, and is the hotspot of the CG algorithm. Hence, we construct a Roofline model to estimate the potential profitability of accelerating this operation with GPUs. We will first focus on fully connected graphs and show later in Section IV how sparsity and locality in the graph can be exploited to further improve performance. Motivated by real-world applications that we encounter as exemplified in the

Fig. 3. A Roofline analysis of the Kronecker product matrix-vector multiplication operation in the conjugate gradient solver for the marginalized graph kernel on the Volta V100 GPU. Hardware metrics are obtained from [7]. Vertical lines correspond to the on-the-fly solver that uses each element $r = 4, 16, 64$ times when computing the matrix-vector product for solving Equation (2) where $E = 0, F = 4, X = 3$.



Appendix of our extended preprint [4], we use an abstract model for the storage and arithmetic cost of the computation where we assume that a floating point number occupies F bytes, an edge label occupies E bytes, and an evaluation of $\kappa_{\mathrm{e}}(\cdot,\cdot)$ costs X floating point operations.

In a naïve implementation, the product matrix $\mathbf{L}_\times \doteq (\mathbf{A} \otimes \mathbf{A}') \odot (\mathbf{E} \overset{\kappa}{\otimes} \mathbf{E}')$ can be precomputed beforehand and reused in the CG loop. Given a pair of graphs each with n and m nodes, respectively, the naïve solver needs to load a floating point matrix $\mathbf{L}_\times$ of $nm \times nm$ elements and a right-hand side vector $\mathbf{p}$ of nm elements, and perform n^2m^2 floating point fused multiply-additions. Hence, the arithmetic intensity of the naïve solver is $2n^2m^2/(n^2m^2F + nmF) \to \frac{2}{F}$, or $\frac{1}{2}$ in single precision mode. On the Volta GPU architecture, the solver is severely memory-bound, achieving at most 3% utilization of the peak floating point performance as predicted by the Roofline plot in Figure 3.

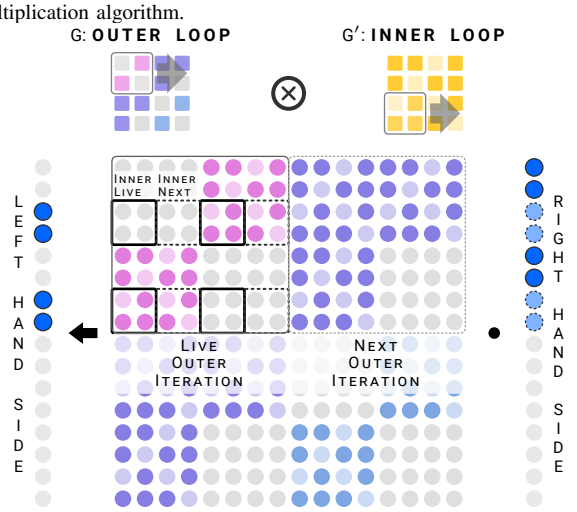
A further disadvantage of the naïve approach is that the product matrix takes up a huge amount of storage space. This could limit both the size of the largest graphs as well as the maximum number of graph pairs that can be simultaneously computed on the GPU.

Algorithm 2 A high-level outline of the on-the-fly Kronecker product matrix-vector multiplication (XMV) algorithm.

```
1 function XMV((A, A'), (E, E'), a, p)
2     streaming length-c chunks in A, E                    OUTER LOOP
3         streaming length-c chunks in A', E'              INNER LOOP
4             for each e_ij in first chunk do
5                 for each e'_i'j' in second chunk do
6                     a_ii' ← a_ii' + κ(e_ij, e'_i'j') · p_jj'
```

In Algorithm 2, we outline an on-the-fly Kronecker product matrix-vector multiplication (XMV) algorithm that directly computes the inner product $\mathbf{L}_\times \cdot \mathbf{p}$, instead of $\mathbf{L}_\times$, in an attempt to trade data movement with arithmetic operations. The algorithm takes advantage of the Kronecker product structure of $\mathbf{L}_\times$ to repeatedly produce the matrix without storing it. This is achieved by streaming elements from the pair of individual graphs and caching them to perform the computation. We can perform c^2 edge kernel evaluations using

Fig. 4. Illustration of the tile-based on-the-fly Kronecker matrix-vector multiplication algorithm.



only fast memory and registers for every pair of length-c chunk of edge weights and labels streamed from the graphs. With a double loop structure, which amortizes the cost of loading one of the two graphs, the on-the-fly approach can achieve an arithmetic intensity of $\frac{c^2X}{c(E+F)} = \frac{cX}{E+F}$, or specifically $\frac{3}{4}c$ in the unlabeled case. Tuning c can thus be used as a straightforward approach to attain the highest utilization of the computing power on the Volta GPU as shown in Figure 3. Note that regenerating the product matrix does not even alter the order of the computational complexity of the matrix-vector multiplication operation, but rather just increases the constant factor. Therefore, this approach is profitable as long as the gain in instruction throughput outweighs the added cost of base kernel evaluations.

III. On-the-Fly Kronecker Product Formation and Matrix-Vector Multiplication

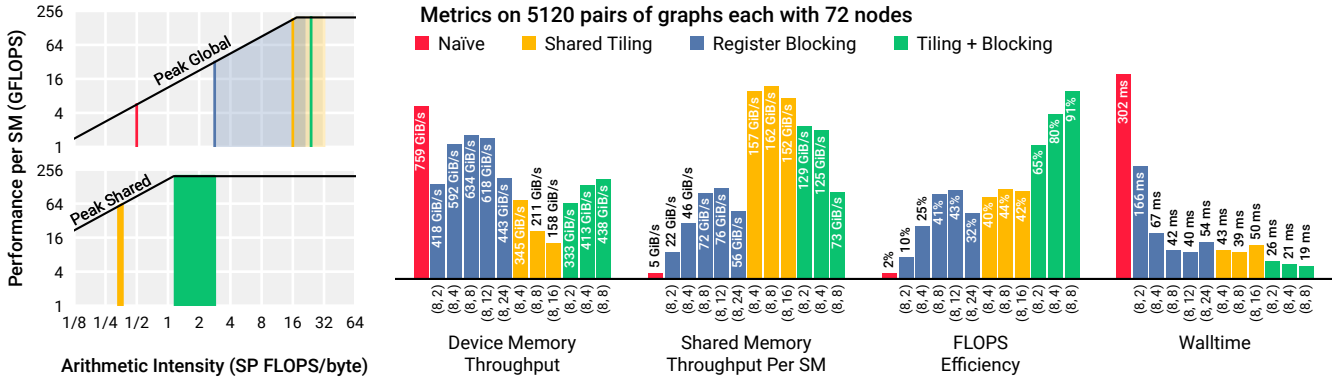
For dense adjacency and edge label matrices of fully connected graphs, we propose three concrete implementations of the on-the-fly XMV algorithm as outlined in Algorithm 2 on the Volta GPU. All three primitives adopt a warp-synchronous programming model, where each warp of 32 consecutive threads work cooperatively on a pair of graphs. A naïve implementation that uses a precomputed product matrix $\mathbf{L}_\times$ is described in our extended preprint [4]. We introduce methods to exploit the sparsity of the graphs in Section IV, and methods for sharing data and work within a thread block in Section V.

We denote the off-chip DDR or HBM memory attached to a GPU as the *device* memory, and the on-chip addressable SRAM as the *shared* memory. A *global* load or store operation accesses the device memory, while a *shared* load or store operation accesses the shared memory.

A. Shared Tiling

Our first implementation, the *shared tiling* primitive, uses the shared memory as a staging area to reduce the usage of global load instructions and device memory bandwidth. Figure 3 shows that it is much easier to approach the theoretical peak

Fig. 5. Detailed benchmark and Roofline analysis of the three on-the-fly XMV primitives. Each primitive other than the naïve one are instantiated with multiple sets of parameters as given underneath each bar. For the *shared tiling* primitive, the two parameters specify the height and width of the tiles that are streamed by the primitive; the two parameters for the *register blocking* primitive have similar meanings to their shared tiling counterparts; for the *tiling & blocking* primitive, the two parameters corresponds to the size of the square tiles streamed and the length of the chunks staged in registers, respectively.



performance on the Volta GPU by loading data from the shared memory with more than 10^4 GB/s peak bandwidth.

As shown in Figure 4, the shared tiling primitive streams $t \times r$ tiles and the corresponding right-hand side elements from the device memory to the shared memory for computation. The tiles and the right-hand side elements are all loaded cooperatively by a warp to ensure coalesced access. The computation on each pair of tiles is parallelized among the threads and across the rows of the product matrix in a round-robin manner. The work is serialized among the columns within each thread. In the actual code, we choose $t = 8$ and explicitly unrolled the row loops by a factor of two to obtain an instruction-level parallelism of two on warps of 32 threads.

B. Register Blocking

Our second implementation, the *register blocking* primitive, uses the register file to stage and reuse matrix elements. While the work is parallelized along the rows in the same way as the shared tiling primitive, each thread here will independently stream length-r chunks from the rows that it owns to compute r^2 elements of the product matrix. Due to the synchronous execution behavior within CUDA thread warps, threads in a warp can still share right-hand side elements via the shared memory because the march across the columns are lock-stepped between consecutive $t \times r$ blocks. The primitive is simpler than the shared tiling primitive, but could theoretically generate more global memory transactions.

C. Combining Shared Tiling and Register Blocking

Our third implementation combines shared tiling and register blocking. This *tiling & blocking* primitive aims to reduce shared and global memory transactions while simultaneously reducing register pressure. Here, $t \times t$ tiles of the adjacency and label matrices are first cached in the shared memory, and then further staged in registers over length-r chunks. This is easily achieved in practice by properly instructing the compiler to unroll the inner loops over column indices.

D. Performance Analysis

From Figure 5, we can see that the tiling & blocking primitive performs the best in terms of time-to-solution. It also achieves the best FLOPS efficiency, defined as the ratio between the actual throughput of floating point operations and the theoretical peak after adjusting for FMA percentage. Hence, the **tiling & blocking primitive with** $t = 8$ **and** $r = 8$ is chosen as the building block for subsequent kernels with more optimizations. We denote the 8×8 square tiles as **octiles** hereafter. The shared tiling primitive and the register blocking primitive performed nearly equally well, yet was not able to achieve the best performance. The shared tiling primitive is unsurprisingly bound by the shared memory throughput as indicated by the measured shared memory bandwidth utilization and the roofline plot. The register blocking primitive is bound by global memory throughput when r is small, yet suffers from register spilling right before it reaches the top of the Roofline model with $r = 24$. Additional tests on a Titan X Pascal graphics card indicate that the shared tiling primitive performs better than the register blocking primitive on accelerator equipped with GDDR memories, but the tiling & blocking primitive still provides the best performance with most balanced utilization of hardware resources.

IV. Explicit Sparsity Exploitation

Many graphs encountered in real-world applications harbor a certain degree of sparsity, which can be exploited to optimize performance. For example, a molecular graph as represented by a SMILES string only contains edges that connect atoms that are chemically bonded, whose number is bound by the maximum number of bonds that an atom can form. A road network graph is also sparse with 3-way and 4-way junctions dominating the map. Even for 3D molecular structures where edges encode pairwise relationships between all atoms, the graphs can still be sparse due to the spatial locality of non-bond interactions.

We adopt a two-level methodology to exploit the sparsity in the graphs. In the first level, we exploit the sparsity observable at the octile granularity and attempt to reduce non-empty tiles through graph reordering. In the second level, we exploit the sparsity within individual octiles using a compact storage scheme that stores only non-zero elements of the tiles and design relevant sparse XMV primitives. In the rest of this

TABLE I

OPERATION COUNT, LOAD/STORE COUNT, AND ASYMPTOTIC ARITHMETIC INTENSITY OF THE ON-THE-FLY KRONECKER PRODUCT MATRIX-VECTOR MULTIPLICATION (XMV) OPERATION ON LINE 10 OF ALGORITHM 1 DURING ONE CG ITERATION. FOR A DETAIL DERIVATION OF THE EQUATIONS, SEE APPENDICES C TO F . n AND m: NUMBERS OF NODES OF THE TWO GRAPHS, RESPECTIVELY; E: BYTE SIZE OF AN EDGE LABEL; F: BYTE SIZE OF AN EDGE WEIGHT; X: BASE KERNEL OPERATION COUNT.

	Naive	$t \times r$ **shared tiling**	$t \times r$ **register blocking**	**length-r register blocking within $t \times t$ shared tiling**
Ops.	$2n^2m^2$	n^2m^2X	n^2m^2X	n^2m^2X
Global Load	n^2m^2F	$n^2m^2(\frac{t}{r}E + \frac{r+t}{r}F)/t^2$	$n^2m^2(\frac{w}{r}E + \frac{w+r}{r}F)/t^2$	$n^2m^2(E+2F)/t^2$
Global Store	nmF	nmF	nmF	nmF
Shared Load	-	$n^2m^2(\frac{r+1}{r}E + \frac{2r+1}{r}F)$	n^2m^2F	$n^2m^2(\frac{r+t}{rt}E + \frac{r+t}{rt}F)$
Shared Store	-	$n^2m^2(\frac{t}{r}E + \frac{r+t}{r}F)/t^2$	n^2m^2F/t^2	$n^2m^2(E+2F)/t^2$
A.I. Global	$\frac{2}{F}$	$\frac{t^2X}{t/rE + (1+t/r)F}$	$\frac{t^2X}{w/rE + (1+w/r)F}$	$\frac{t^2X}{E+2F}$
A.I. shared	-	$\frac{X}{(1+1/r)E + (2+1/r)F}$	$\frac{X}{(1+1/t^2)F}$	$\frac{X}{(1/r+1/t)E + (1/r+1/t)F}$

section, we use graph and matrix terms interchangeably.

A. Inter-Tile Sparsity

The sparsity of graphs can be readily exploited within the on-the-fly XMV framework by discarding empty tiles that contain no edges. This pruning process can be easily implemented as a pre-processing pass on either the CPU or the GPU, but its efficiency depends on the probability of finding empty tiles. Hence, we resort to reordering algorithms to group the nonzeros of the matrix into as few tiles as possible. Among a plethora of heuristics in the literature for reordering matrices, the ones that we have experimented with are:

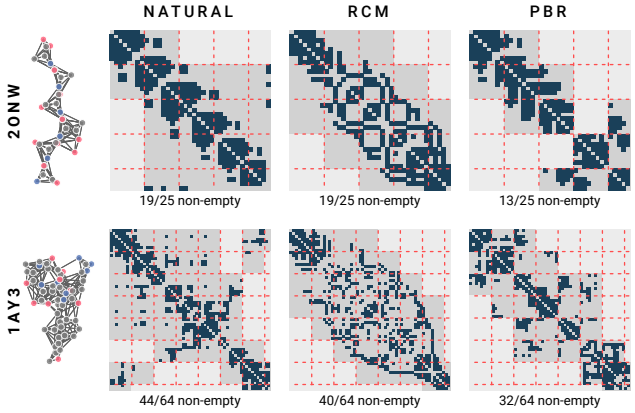
- a custom partitioning-based reordering (PBR) algorithm [8] that specifically targets the objective of minimizing the number of non-empty tiles;
- the Reverse Cuthill-McKee (RCM) algorithm [9], which is a heuristic that is widely used for fill-in and matrix bandwidth reduction;
- a scheme based on solving the Traveling Salesman Problem (TSP) [10] with heuristics.
- a scheme using space-filling curves such as the Morton curve [11] or the Hilbert curve when the vertices are known to be embedded in a Euclidean space.

Among the four reordering methods, we have found that the PBR-based method delivers the most reduction in non-empty octiles using a moderate amount of time.

The Morton-based method delivers less reduction than RCM and PBR despite being marginally faster. The TSP-based method achieves a reduction rate between RCM and PBR. However, the running time of TSP-based reordering algorithm is substantially longer than all other reordering methods by orders of magnitude. Hence, we decide to focus only on RCM and PBR in subsequent discussions, and present two examples of molecular graphs representing the protein `2ONW` and `1AY3` from the Protein Data Bank (PDB) in their natural orders, the RCM order and the PBR order, respectively, in Fig. 6.

In cases such as when the graphs represent 3D protein structures, the nodes in their *natural* order, *e.g.* the order of the corresponding amino acid residues in the primary structure of the protein, can already be a good ordering of the nodes. However, reordering in general can be very useful because the natural order of the graphs are rarely available. Moreover, the PBR order can beat the natural order in reducing non-empty tiles for different datasets as evident in Figures 6 and 7.

Fig. 6. Natural order of two graphs from the PDB dataset (Protein Crystal Structure) and their reordered versions with the RCM and PBR methods.



Partitioning-based Reordering (PBR) for improved tile density

The goal of PBR in our case is to reorder nodes in a graph $G = (V, E)$, *i.e.*, to come up with a permutation of the rows and columns of the corresponding matrix, so that the number of non-empty square $t \times t$ tiles is minimized.

Let $\Pi(G) = \{V_1, V_2, \ldots, V_K\}$ be a perfectly balanced K-way vertex partition of G with $K = \lceil |V|/t \rceil$, where all parts in $\Pi(G)$ with the possible exception of the last part has exactly the same number of vertices. $\Pi(G)$ then implies a vertex ordering, where the vertices in V_k are ordered before the vertices in V_{k+1}, for $1 \leq k < K$. Observe that for any $1 \leq k \neq \ell \leq K$, if there is at least one edge between the nodes within V_k and V_ℓ, then the tile at the intersection of kth row stripe and ℓth column stripe of the matrix, as well as its symmetric counterpart, are non-empty. If there are no

edges between V_k and V_ℓ, then the respective tiles are empty. Therefore, we can define the objective of PBR as finding the $\Pi(G)$ that minimizes

$$|\{(V_k, V_\ell) : k \neq \ell \text{ and } (v_i \in V_k, v_j \in V_\ell) \in E\}|. \quad (3)$$

To seek a good $\Pi(G)$, we utilize an approach [8] that is fast and has a consistent objective with (3), but does not always guarantee a perfectly balanced partition. Perfectly balanced graph partitioning problem has previously been studied in the literature [12], [13], usually with a different objective of minimizing the number of inter-partition edges. These approaches also emphasize partition quality over speed and rely on expensive algorithms such as tabu search. The approach that we use here is based on a recursive bipartitioning scheme that was originally proposed for reducing message counts in parallel applications, which can also be modeled as off-diagonal blocks in a matrix. The bipartitioning heuristics are much faster than the algorithms used for perfectly balanced partitioning.

Even though the PBR algorithm in [8] does not guarantee perfectly balanced partitions, imbalance is a rare problem when all vertices have the same weight, which is exactly the case in our work. Nonetheless, for the cases in which the partitioner could not obtain a perfectly balanced partition, we append an extra refinement step to move vertices from the overloaded part to the underloaded part based on the Fiduccia-Mattheyses (FM) algorithm [14]. We also utilized a custom weight distribution, as opposed to a single imbalance parameter, in the recursive bipartitioning process to promote equally sized parts.

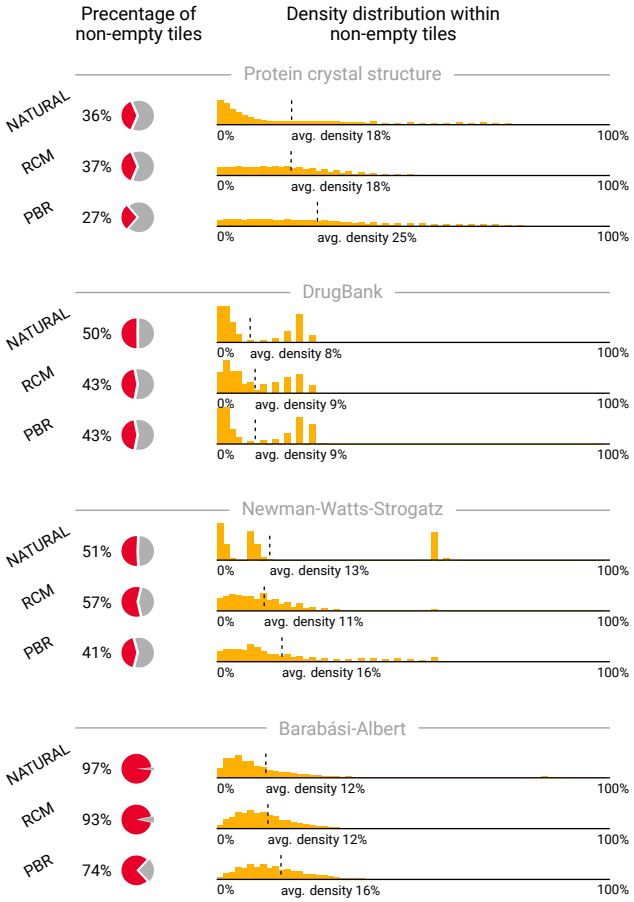
Finally, we adjust the parameters of the partitioner to ensure a tight balancing constraint by setting the refinement algorithm to boundary FM with tight balance.

In addition, we set the cost of the message nets, a parameter that emphasizes the importance of the reduction of non-empty tiles, to a large value such as 50.

In Fig. 7, we illustrate the performance of the PBR order as compared to the natural order and the RCM order on four different datasets which are detailed in Section VI. PBR is able to achieve the best reduction over the natural ordering in all datasets, while RCM can only improve the non-empty octile count in two of the datasets.

Reordering overhead Reordering is justified when its cost is smaller than the computational savings it enables. The PBR reordering incurs a linear-time pre-processing overhead proportional to the number of non-zeros in the matrices, while the marginalized graph kernel incurs a quadratic cost with regard to the number of non-zeros during each CG iteration. Moreover, the graph kernel often has to be evaluated on all pairs of graphs for hundreds of times to train a machine learning model, while the training data need only to be reordered once. Hence, the overhead of the PBR reordering can be easily amortized.

Fig. 7. Analysis of the sparsity pattern in four datasets of real and synthetic graphs after reordering.



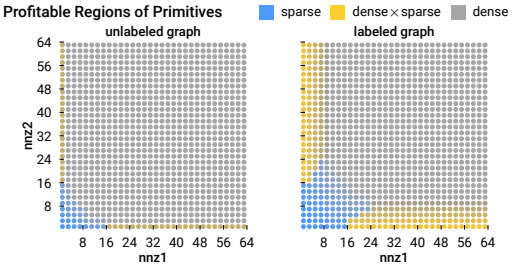
B. Intra-Tile Sparsity

As demonstrated in Section III, the tiling & blocking kernel is very efficient for computing between dense tiles on the GPUs. In fact, the kernel can still be very efficient on sparse graphs, because our reordering algorithms tend to create locally dense areas in the matrices. Nonetheless, as seen from Figure 7, although the reordering methods indeed increase the octile density compared to the natural order, the non-empty tiles can still be up to 90% empty. Hence, we can attain considerable savings by storing and processing only the nonzero elements instead of treating the tiles as dense.

In order to exploit sparsity within an octile, we use a compact layout to store only nonzero elements. An accompanying 64-bit integer, whose ith element is set if the ith element is nonzero, is used to locate the nonzero elements in the original octile. We then rely on bit manipulations to find the indices of the nonzero elements. Compared to the dense octile representation, the sparse representation reduces unnecessary global memory transactions besides wasting flops. However, this comes at the expense of increased shared memory utilization.

Hybrid dense-sparse computation Given a pair of tiles, the optimal way of evaluating their XMV operation depends on the sparsity of both tiles. As such, we designed two new types of XMV primitives in addition to the $\text{dense} \times \text{dense}$ kernel: (i)

Fig. 8. Microbenchmark on performance of the three types of dense/sparse primitives.



a primitive for multiplying a dense and a sparse tiles, or vice versa (dense$\times$sparse), and (ii) a primitive for multiplying two sparse tiles (sparse $\times$ sparse).

Utilizing a single primitive for the entire execution may hurt the performance as the kernels' performance largely depend on the octile density, which can vary significantly within and across datasets as visualized in Figure 7.

Fig. 8 illustrates the best performing product kernel for varying number of nonzeros of the two source octiles for both labeled and unlabeled graphs.

The sparse $\times$ sparse kernel performs the best when each of the octiles contain up to 8-10 nonzeros for the unlabeled graphs and up to 16 nonzeros for the labeled graphs.

The dense$\times$dense kernel runs the fastest from that point on as both of the octiles get denser.

In the rest, the dense$\times$sparse kernel performs better.

In our production kernel, we dynamically select either the sparse$\times$sparse or the dense$\times$dense kernel prior to each octile product depending on the type of the graph and the number of products the two octiles require.

The octiles are always stored in a compact form and expanded in the shared memory after loading them from the global memory.

V. Tile Sharing and Load Balancing

A. Block-Level Sharing

To fully utilize the GPU, which can simultaneously execute thousands of warps on the fly, we can perform the graph kernel computations between many different pairs of graphs simultaneously within a single kernel launch.

One option is to assign each thread warp a unique graph pair. This approach resembles a SIMD programming model while no synchronization and/or cooperation between the thread warps is needed. It is unfavorable when low-latency computations on a few graphs are required because a small and fixed number of threads are allocated for each pair of graphs. On a Volta GPU, thousands of graph pairs are needed to provide enough parallelism to saturate the instruction pipeline.

A second option is to further parallelize the computation within a thread block, whose size can vary between 32 to 1024 threads on CUDA GPUs. A first and obvious benefit of this approach is that it provides us the ability to use block size to adjust the latency for computing each pair of graphs, as well as allowing a smaller number of graph pairs to saturate the entire GPU. This block-based cooperative approach also has the potential to further improve performance by allowing warps within a block to share the octiles in shared memory. As revealed in the Roofline analysis, larger tiles results in more data reuse, less redundant load/store operations, and higher arithmetic intensity. However, there is a limit on the size of tiles that a warp can hold without constraining occupancy, *i.e.* the number of warps on the fly. To work around this, we let all the N warps in a CUDA thread block each load an octile, and then share the octiles to compute N^2 octile Kronecker matrix vector multiplications.

Tile sharing requires block-level synchronization before and after octile loading. Besides, atomic accumulations are necessary for writing to the output vector since the COO storage format obscures the effort to schedule workload among the warps in ways such that the output could be conflict-free. However, the performance impact on CUDA GPUs should be very minimal because atomic accumulations whose outputs are not immediately used are carried out by nonblocking atomic reduction instructions. As such, the threads that commit the atomic accumulations will not get stalled.

B. Inter-Block Load Balancing

Thanks to the independence of the computations between different pairs of graphs, load balancing is relatively straightforward since tasks can be easily relocated across computing units. Aside from transient factors such as warp scheduling, cache conflict, and atomics, the primary source of load imbalance is the variation of graph size and sparsity pattern that affect the problem size as well as the number of conjugate gradient iterations for convergence.

VI. Benchmark Dataset

A. Synthetic Graphs

To test the performance of our solver, the Newman-Watts-Strogatz (NWS) algorithm and the Barabási-Albert (BA) algorithm were used to generate synthetic graphs of small-world and scale-free characteristics, respectively.

B. Real-World Dataset

The graph kernel is further tested on real-world datasets as summarized below:

1) The PDB-3k dataset is a 1324-structures subset of the Protein Data Bank database [15] containing proteins less than 3000 in weight and contain no DNA/RNA complexes. Each protein is converted into a graph with nodes representing heavy atoms. An adjacency rule is used to create edges between spatially neighboring atoms such that the weights smoothly decay to zero at a certain cutoff distance. The edges are labeled with the interatomic distance between its endpoints.
2) DrugBank [16] is a comprehensive database containing information about drug molecules. The dataset contains more than 10^4 drug molecules, 10607 of which has a corresponding linearized representation as a SMILES string, which is essentially obtained by a depth-first traversal of a chemical graph. A rich body of node and edge

attributes can be extracted from the SMILES strings such as hybridization state, charge, bond order, and conjugacy.

VII. Performance Measurement and Analysis

Benchmarks are performed on the Summit supercomputer at Oak Ridge National Laboratory. The runtime and performance metrics of our GPU kernels are measured using the `nvprof` program from the CUDA Toolkit, while CPU-side time measurements are obtained using the `time.perf_counter_ns()` method from the Python standard library.

A. Performance Improvement of Proposed Optimization Techniques

In this section, we characterize and compare the performance gain enabled by the previously described optimization techniques on both the synthetic and real-world graph datasets. The measurements are carried out using the naïve kernel as a baseline and then enabling the optimization techniques one at a time in the same order as they are introduced in the text. For the synthetic graph datasets, we generate 160 graphs containing 96 nodes for each type with the following parameters:

- Newman-Watts-Strogatz: $k = 3$, $p = 0.1$;
- Barabási-Albert: $m = 6$.

We also test the kernel on all graphs in the PDB and DrugBank datasets.

From Figure 9, we can see that the performance improvements brought about by the techniques depend on the characteristics of the actual dataset. Overall, the speedup is more impressive on the real-world datasets which contains larger and more diverse graphs. It turns out inter-tile sparsity exploitation, when directly applied on the graphs in their natural order, is effective for all datasets except for the scale-free networks which contain poor locality. PBR-based reordering performs very well and increased the performance of the solver on all datasets. The adaptive dense/sparse primitive switch and the compact tile storage format can improve solver performance on all datasets.

In contrast, block-level tile sharing leads to a significant performance improvements on DrugBank and mild improvements on the others. This is caused by the large size variation — between 1 and 551 — in the DrugBank dataset that costs a long time for a single warp to compute a pair of the largest molecules. Dynamic scheduling brings about marginal performance improvements because the GPUs are fully saturated by the large datasets anyway.

B. Performance Comparison with State-of-the-Art Packages

We further compare the performance of our solver against two state-of-the-art packages for graph kernel computations: GraKeL [17] and GraphKernels [18]. GraKeL is a Python package compatible with scikit-learn [19]. The compute-intesive part of GraKeL is implemented using Cython [20], which compiles codes written in a Python-like syntax into binaries on the target machine. The GraphKernels package is implemented in C++ and has a Python frontend generated with SWIG [21]. Both packages run only on CPUs, although GraKeL does support parallelism using multiple processes but with limited scaling efficiency. When executing the codes on the Power9 cores of Summit, we allocate 4 physical cores to GraKeL and 1 physical core to GraphKernels.

As shown by Figure 10, our solver significantly outperforms both GraKeL and GraphKernels by 3-4 orders of magnitude on real-world datasets. Besides performance, it is worth noting that we had to carry out the computation using a relative large stopping probability for both GraKeL and GraphKernels to avoid convergence failures. Coincidentally, a larger stopping probability can reduce time to solution at the expense of the discriminating power of the kernel. Our presented kernel does not have a convergence issue and can compute using stopping probability values as small as 0.0005.

Fig. 9. Time to solution for the presented solver equipped with different optimization techniques (data with an asterisk are projected from ensembles of 32 random subsets of the entire dataset). Within each dataset, each bar repesents a kernel that incorporates an optimization techniques while inheriting everything else from the kernel below. Label interpretation: `Dense` – the naïve kernel, `Sparse` – sparsity explotation at the inter-tile level, `+Reorder` – enabling PBR graph reordering, `+Adaptive` – adaptive switching between dense and sparse tile primitives, `+Compact` – compact storage format of each tile, `+Block`: sharing of tiles within a block, `+DynSched` – dynamic work scheduling.

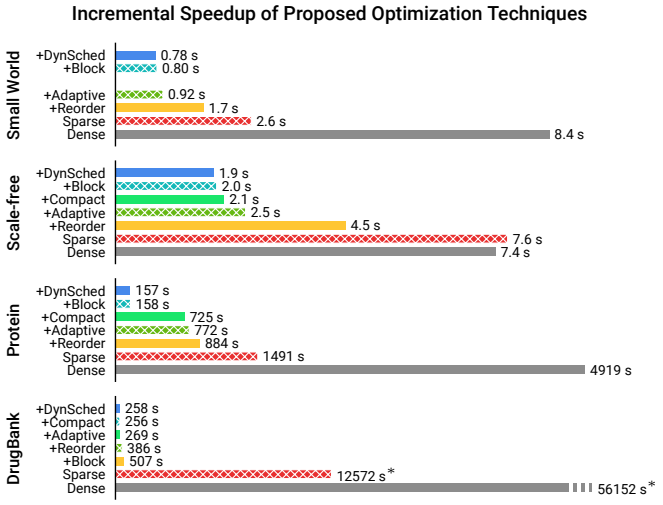
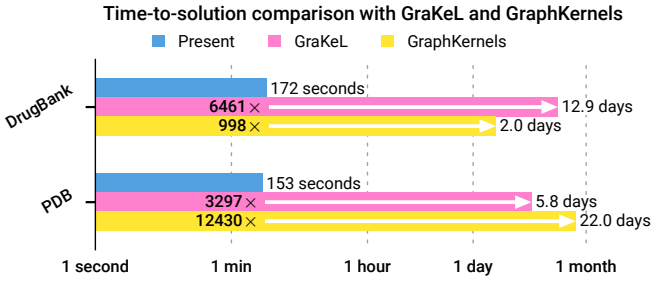


Fig. 10. Performance comparison of the presented solver with existing packages.



VIII. Related Work

The two packages GraKeL and Graph-Kernels that we have compared against in section VII-B provide the closest func-

tionality to our solver, but we significantly outperform them by several orders of magnitude. Moreover, only GraKeL supports graphs with both labeled vertices and labeled edges, which have been proven to be crucial for building accurate machine learning models for molecular systems [2]. It is easily verifiable that the normalized Gramian matrix generated using unlabeled graphs essentially contains numbers all very close to unity, implying that all graphs are identical to each other under the unlabeled similarity measure. As such, we believe the present solver, which can efficiently compute the graph kernel for labeled graphs, represents not only an improvement in terms of computational speed, but also an enhancemennt of the functionality available to the end users.

The graph kernel is fundamentally different from network alignment algorithms [22] that can provide an estimate on the similarity of two graphs. There are two issues that make network alignment algorithms unsuitable as kernels between graphs: 1) in general, they do not define a proper metric that respects triangle inequality in some feature space, *i.e.* they are not positive definite functions; and 2) they are potentially more expensive as the alignment operation typically performs more work in addition to computing nodal similarities.

Last, the work of Livi *et al.* [23] contains an algorithmic motif concerning parallel graph tensor product calculations for inexact graph matching, while their formulation also has a product weight matrix that is computed using a vertex kernel and an edge kernel. However, the product graph is not used to construct a linear system that has to be solved.

IX. Conclusion

In this paper, we presented a series of algorithms to accelerate a marginalized graph kernel solver on GPU. The solver is essentially an implementation of the conjugate gradient method for a generalized Laplacian system induced by the Kronecker product of a pair of graphs. Via roofline analysis, we identified that the solver will likely be memory-bound due to the matrix-vector inner product operation in the CG method. We overcame this issue by taking advantage of the Kronecker product structure of the system. In our approach, we do not precompute the product linear system, but rather stream and cache the original graph pair in tiles, and compute the product system on-the-fly. This approach significantly reduces global memory traffic, and only increases the asymptotic arithmetic operation count by a constant factor, which can be easily offset by the large gain in instruction throughput. Moreover, the solver can take advantage of the sparsity in the graph by making use of a two-level storage format to trim out zero elements. We compared the performance of the solver with existing packages and demonstrated that the present implementation can deliver significant speedups.

This work also exemplifies the paradigm of applying linear algebra concepts and techniques to solving graph problems. The graph kernel problem constitutes a concrete example of the need for standardized application programming interfaces for graph tensor products in specifications such as GraphBLAS [24], and prompt for the development of high-performance and general implementations of the interface. Our work suggests that the semantics for the *inner product* between tensor product structures may see wider applicability than that for the mere computation of the tensor product itself.

Acknowledgment

This work was supported by the Luis W. Alvarez Postdoctoral Fellowship at Lawrence Berkeley National Laboratory. This work is also supported in part by the Applied Mathematics program of the DOE Office of Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231, and in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. DOE Office of Science and the NNSA. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725. YHT thanks David Williams-Young, and Caitlin A Whitter for helpful discussions and suggestions.

References

[1] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. AAAI Press, 2003, pp. 321–328, 00000.

[2] Y.-H. Tang and W. A. de Jong, "Prediction of atomization energy using graph kernel and active learning," *The Journal of Chemical Physics*, vol. 150, no. 4, p. 044107, Jan. 2019, autocitation-1.

[3] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, no. suppl_1, pp. i47–i56, Jun. 2005.

[4] Y.-H. Tang, O. Selvitopi, D. Popovici, and A. Buluç, "A High-Throughput Solver for Marginalized Graph Kernels on GPU," Oct. 2019.

[5] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, vol. 11, no. Apr, pp. 1201–1242, 2010, 00000.

[6] S. Vishwanathan, K. M. Borgwardt, and N. N. Schraudolph, "Fast Computation of Graph Kernels," in *NIPS*, vol. 19, 2006, pp. 131–138.

[7] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *arXiv:1804.06826 [cs]*, Apr. 2018.

[8] O. Selvitopi, S. Acer, and C. Aykanat, "A recursive hypergraph bipartitioning framework for reducing bandwidth and latency costs simultaneously," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 345–358, Feb. 2017.

[9] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, ser. Prentice-Hall Series in Computational Mathematics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[10] A. Pinar and M. T. Heath, "Improving Performance of Sparse Matrix-Vector Multiplication," in *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Nov. 1999, pp. 30–30.

[11] Y.-H. Tang and G. E. Karniadakis, "Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications," *Computer Physics Communications*, vol. 185, no. 11, pp. 2809–2822, Nov. 2014, 00025.

[12] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, Eds. Springer Berlin Heidelberg, 2013, pp. 164–175.

[13] U. Benlic and J.-K. Hao, "An effective multilevel tabu search approach for balanced graph partitioning," *Comput. Oper. Res.*, vol. 38, no. 7, pp. 1066–1075, Jul. 2011.

[14] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, ser. DAC '82. Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181.

[15] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The Protein Data Bank," *Nucleic Acids Research*, vol. 28, no. 1, pp. 235–242, Jan. 2000.

[16] D. S. Wishart, Y. D. Feunang, A. C. Guo, E. J. Lo, A. Marcu, J. R. Grant, T. Sajed, D. Johnson, C. Li, Z. Sayeeda, N. Assempour, I. Iynkkaran, Y. Liu, A. Maciejewski, N. Gale, A. Wilson, L. Chin, R. Cummings, D. Le, A. Pon, C. Knox, and M. Wilson, "DrugBank 5.0: A major update to the DrugBank database for 2018," *Nucleic Acids Research*, vol. 46, no. D1, pp. D1074–D1082, Jan. 2018.

[17] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgianis, "GraKeL: A Graph Kernel Library in Python," Jun. 2018.

[18] M. Sugiyama, M. E. Ghisu, F. Llinares-López, and K. Borgwardt, "Graphkernels: R and Python packages for graph comparison," *Bioinformatics*, vol. 34, no. 3, pp. 530–532, Feb. 2018.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, Oct. 2011.

[20] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science and Engg.*, vol. 13, no. 2, pp. 31–39, Mar. 2011.

[21] D. M. Beazley, "Automated Scientific Software Scripting with SWIG," *Future Gener. Comput. Syst.*, vol. 19, no. 5, pp. 599–609, Jul. 2003.

[22] R. Singh, J. Xu, and B. Berger, "Global alignment of multiple protein interaction networks with application to functional orthology detection," *Proceedings of the National Academy of Sciences*, vol. 105, no. 35, pp. 12 763–12 768, Sep. 2008.

[23] L. Livi and A. Rizzi, "Parallel algorithms for tensor product-based inexact graph matching," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, Jun. 2012, pp. 1–8.

[24] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 643–652.

[25] Wendland, *Scattered Data Approximation*. Cambridge university press, 2004, vol. 17.

[26] Y.-H. Tang, D. Zhang, and G. E. Karniadakis, "An atomistic fingerprint algorithm for learning ab initio molecular force fields," *The Journal of Chemical Physics*, vol. 148, no. 3, p. 034101, Jan. 2018, 00000.

APPENDIX

A. Derivation of the Linear Algebra Form of the Marginalized Graph Kernel

The marginalized graph kernel in its raw form reads:

$$K(G,G') = \sum_{\ell=1}^{\infty} \sum_{\mathbf{h}} \sum_{\mathbf{h}'} \Bigg[p_s(h_1)\ p'_s(h'_1)\ \kappa_{\mathrm{v}}(v_{h_1}, v'_{h'_1})\ p_q(h_\ell)\ p'_q(h'_\ell)\ \left(\prod_{i=2}^{\ell} p_t(h_i|h_{i-1})\right)\ \left(\prod_{j=2}^{\ell} p'_t(h'_j|h'_{j-1})\right) \left(\prod_{k=2}^{\ell} \kappa_{\mathrm{v}}(v_{h_k}, v'_{h'_k}) \kappa_{\mathrm{e}}(e_{h_{k-1}h_k}, e'_{h'_{k-1}h'_k})\right) \Bigg]. \tag{4}$$

Even though the random walk description permits an intuitive interpretation, the marginalized graph kernel could be much more efficiently computed by transforming the task into an equivalent one that involves solving a linear system which resembles a graph Laplacian that is defined on the Kronecker product of the pair of graphs [2].

In fact, Equation (4) can be reformulated under the spirit of dynamic programming as [1]:

$$K(G,G') = \sum_{h_1 \in V, h'_1 \in V'} p_s(h_1)\ p'_s(h'_1)\ \kappa_{\mathrm{v}}(h_1, h'_1)\ R_\infty(h_1, h'_1), \tag{5}$$

where R_∞ is the solution to the linear system:

$$R_\infty(h_1, h'_1) = p_q(h_1)\ p'_q(h'_1) + \sum_{i \in V, j \in V'} t(i,j,h_1,h'_1)\ R_\infty(i,j), \tag{6}$$

with

$$t(i,j,h_1,h'_1) \coloneqq p_t(i|h_1)\ p'_t(j|h'_1)\ \kappa_{\mathrm{v}}(v_i, v_j)\ \kappa_{\mathrm{e}}(e_{i\,h_1}, e_{j\,h'_1}). \tag{7}$$

Equations (5) to (7) exhibit a Kronecker product structure, which can be readily recognized in matrix form:

$$K(G,G') = \left(\mathbf{p} \otimes \mathbf{p}'\right)^{\mathsf{T}} \cdot \mathbf{diag}\left(\mathbf{v} \overset{\kappa_{\mathrm{v}}}{\otimes} \mathbf{v}'\right) \cdot \mathbf{r}_\infty, \tag{8}$$

with $\mathbf{r}_\infty$ being the solution to the linear system

$$\mathbf{r}_\infty = \mathbf{q} \otimes \mathbf{q}' + \left[\left(\mathbf{P} \otimes \mathbf{P}'\right) \odot \left(\mathbf{E} \overset{\kappa_{\mathrm{e}}}{\otimes} \mathbf{E}'\right)\right] \cdot \mathbf{diag}\left(\mathbf{v} \overset{\kappa_{\mathrm{v}}}{\otimes} \mathbf{v}'\right) \cdot \mathbf{r}_\infty, \tag{9}$$

where

$\mathbf{v}$	is the vertex label vector of G with $\mathbf{v}_i = v_i$;
$\mathbf{p}$	is the starting probability vector of G with $\mathbf{p}_i = p_s(v_i)$;
$\mathbf{q}$	is the stopping probability vector of G with $\mathbf{q}_i = p_q(v_i)$;
$\mathbf{P}$	is the transition probability matrix of G defined as $\mathbf{D}^{-1}\mathbf{A}$;
$\mathbf{E}$	is the edge label matrix of G with $\mathbf{E}_{ij} = e_{ij}$;
$\mathbf{v}'$, $\mathbf{p}'$, $\mathbf{q}'$, $\mathbf{P}'$, $\mathbf{E}'$	are the corresponding vectors and matrices for G';
$\overset{\kappa_{\mathrm{v}}}{\otimes}$	is the generalized Kronecker product between $\mathbf{v}$ and $\mathbf{v}'$ with respect to κ_{v};
$\overset{\kappa_{\mathrm{e}}}{\otimes}$	is the generalized Kronecker product between $\mathbf{E}$ and $\mathbf{E}'$ with respect to κ_{e}.

For clarity of the discussion below, we denote

$$\begin{aligned}
&\mathbf{V}_\times \coloneqq \mathbf{diag}\left(\mathbf{v} \overset{\kappa_{\mathrm{v}}}{\otimes} \mathbf{v}'\right),\\
&\mathbf{D}_\times \coloneqq \mathbf{diag}(\mathbf{d}) \otimes \mathbf{diag}(\mathbf{d}'),\\
&\mathbf{A}_\times \coloneqq \mathbf{A} \otimes \mathbf{A}',\\
&\mathbf{P}_\times \coloneqq \mathbf{P} \otimes \mathbf{P}' = \mathbf{D}_\times^{-1}\mathbf{A}_\times,\\
&\mathbf{E}_\times \coloneqq \mathbf{E} \overset{\kappa_{\mathrm{e}}}{\otimes} \mathbf{E}',\\
&\mathbf{p}_\times \coloneqq \mathbf{p} \otimes \mathbf{p}',\\
&\mathbf{q}_\times \coloneqq \mathbf{q} \otimes \mathbf{q}'.
\end{aligned}$$

To solve eq. (9), first observe that only the product $\mathbf{V}_\times \mathbf{r}_\infty$ as a whole is needed to compute $K(G,G')$. We can thus rearrange

eq. (9) to form a symmetric linear system.

$$\mathbf{r}_\infty - (\mathbf{P}_\times \odot \mathbf{E}_\times)\,\mathbf{V}_\times \mathbf{r}_\infty = \mathbf{q}_\times, \tag{10}$$

$$\left(\mathbf{V}_\times^{-1} - \mathbf{P}_\times \odot \mathbf{E}_\times\right)\mathbf{V}_\times \mathbf{r}_\infty = \mathbf{q}_\times, \tag{11}$$

$$\mathbf{V}_\times \mathbf{r}_\infty = \left(\mathbf{V}_\times^{-1} - \mathbf{P}_\times \odot \mathbf{E}_\times\right)^{-1} \mathbf{q}_\times \tag{12}$$

$$= \left[\mathbf{V}_\times^{-1} - \left(\mathbf{D}_\times^{-1}\mathbf{A}_\times\right) \odot \mathbf{E}_\times\right]^{-1} \mathbf{q}_\times \tag{13}$$

$$= \left(\mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times\right)^{-1} \mathbf{D}_\times \mathbf{q}_\times. \tag{14}$$

The linear system $\mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times$ in eq. (14) is symmetric and positive-definite, as long as $q > 0$, $\kappa_\mathrm{v}(\cdot,\cdot) <= 1$, and $\kappa_\mathrm{e}(\cdot,\cdot) <= 1$. Thus, we have reached the full expression for the marginalized graph kernel in matrix form, the solution of which is the central focus of this paper:

$$K(G, G') = \mathbf{p}_\times^\mathsf{T} \left(\mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times\right)^{-1} \mathbf{D}_\times \mathbf{q}_\times. \tag{15}$$

B. Example of edge label and kernel

Some examples of edge kernels used in practice are: 1) a square exponential kernel $\kappa^{\mathrm{SE}}(e_1, e_2) \doteq \exp[-\alpha(e_1 - e_2)^2]$ consumes two floats and carries out 3 multiplication and 1 exponentiation; 2) a degree n compact polynomial radial basis kernel, *e.g.* in the form $\kappa^{\mathrm{P}}(e_1, e_2) \doteq \sum_i \alpha_i (e_1 - e_2)^i$ [25], [26] consumes two floats and performs n chained FMA instructions; 3) a Kronecker product kernel $\kappa^{\mathrm{kron}}(e_1, e_2) \doteq \prod_i \kappa_i(e_1^i, e_2^i)$ consumes $2n$ inputs and carry out a linearly proportional number of operations; 4) an R-convolutional kernel $\kappa^{\mathrm{R}}(e_1, e_2) = \sum_i \sum_j \kappa(e_1^i, e_2^j)$ consumes $2n$ inputs and carry out a quadratically proportional number of arithmetics.

C. Pseudocode, I/O and Operation Counts of the Naïve Approach

For explicit (full-matrix) mode, where we precompute $\mathbf{L} \doteq \mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times$, it is trivial to determine that $n^2m^2F + nmF$ bytes will have to be loaded from global memory,

```
1  parfor i ∈ [0 ... nm] do
2      a_i ← 0
3      for J ∈ [0, WARPSIZE, ... nm] do
4          GLOBALLOAD(p_{J+LANE})                          LD.G F
5          for j ∈ J + [0, 32) do
6              GLOBALLOAD(L_ij)                            LD.G F
7              SHUFFLE(p_j) from lane j
8              a_i ← a_i + L_ij × p_j                      OPS 2
9      GLOBALSTORE(a_i)                                    ST.G F
```

line	unit cost	iteration count	subtotal
global load $(n, m \to \infty)$			$\boldsymbol{\approx n^2m^2F}$
4	F	$n^2m^2/32$	$n^2m^2F/32$
6	F	n^2m^2	n^2m^2F
global store $(n, m \to \infty)$			$\boldsymbol{nmF}$
9	F	nm	nmF
Operations $(n, m \to \infty)$			$\boldsymbol{2n^2m^2}$
8	2	n^2m^2	$2n^2m^2$

D. Pseudocode, I/O and Operation Counts of the Shared Tiling Primitive

```
1  for I ∈ [0, t, 2t ... n), I' ∈ [0, t, 2t, ... m) do
2      parfor i ∈ I + [0, t), i' ∈ I' + [0, t) do
3          a_{ii'} ← 0
4      for J ∈ [0, r, 2r ... n) do
```

5 GLOBALLOAD($\mathbf{A}_{I+[0,t),J+[0,r)}$) LD.G rtF

6 SHAREDSTORE($\mathbf{A}_{I+[0,t),J+[0,r)}$) ST.S rtF

7 GLOBALLOAD($\mathbf{E}_{I+[0,t),J+[0,r)}$) LD.G rtE

8 SHAREDSTORE($\mathbf{E}_{I+[0,t),J+[0,r)}$) ST.S rtE

9 **for** $J' \in [0, r, 2r \dots m)$ **do**

10 GLOBALLOAD($\mathbf{A}'_{I'+[0,t),J'+[0,r)}$) LD.G rtF

11 SHAREDSTORE($\mathbf{A}'_{I'+[0,t),J'+[0,r)}$) ST.S rtF

12 GLOBALLOAD($\mathbf{E}'_{I'+[0,t),J'+[0,r)}$) LD.G rtE

13 SHAREDSTORE($\mathbf{E}'_{I'+[0,t),J'+[0,r)}$) ST.S rtE

14 GLOBALLOAD($\mathbf{p}_{J+[0,r),J'+[0,r)}$) LD.G r^2F

15 SHAREDSTORE($\mathbf{p}_{J+[0,r),J'+[0,r)}$) ST.S r^2F

16 **parfor** $i \in I + [0, t), i' \in I' + [0, t)$ **do**

17 **for** $j \in J + [0, r)$ **do**

18 SHAREDLOAD($\mathbf{A}_{ij}, \mathbf{E}_{ij}$) LD.S $E + F$

19 **for** $j' \in J' + [0, r)$ **do**

20 SHAREDLOAD($\mathbf{A}_{i'j'}$) LD.S F

21 SHAREDLOAD($\mathbf{E}_{i'j'}$) LD.S E

22 SHAREDLOAD($\mathbf{p}_{jj'}$) LD.S F

23 $\mathbf{a}_{ii'} \leftarrow \mathbf{a}_{ii'} + \mathbf{p}_{jj'} \cdot \mathbf{A}_{ij} \cdot \mathbf{A}'_{i'j'}$ $\cdot\, \kappa_{\mathrm{e}}(\mathbf{E}_{ij}, \mathbf{E}'_{i'j'})$ OPS X

24 GLOBALSTORE($\mathbf{a}_{I+[0,t),I'+[0,t)}$) ST.G t^2F

Thus, total memory load and arithmetic operations for shared tiling are summarized as below:

line	unit cost	iteration count	subtotal
global load ($n, m \to \infty$)			$\boldsymbol{n^2m^2(\frac{t}{r}E + \frac{r+t}{r}F)/t^2}$
5	rtF	n^2m/rt^2	n^2mF/t
7	rtE	n^2m/rt^2	n^2mE/t
10	rtF	n^2m^2/r^2t^2	n^2m^2F/rt
12	rtE	n^2m^2/r^2t^2	n^2m^2E/rt
14	t^2F	n^2m^2/t^4	n^2m^2F/t^2
global store ($n, m \to \infty$)			$\boldsymbol{nmF}$
24	t^2F	nm/t^2	nmF
shared load ($n, m \to \infty$)			$\boldsymbol{n^2m^2(\frac{r+1}{r}E + \frac{2r+1}{r}F)}$
18	$E + F$	n^2m^2/r	$n^2m^2(E + F)/r$
20	F	n^2m^2	n^2m^2F
21	F	n^2m^2	n^2m^2E
22	F	n^2m^2	n^2m^2F
shared store ($n, m \to \infty$)			$\boldsymbol{n^2m^2(\frac{t}{r}E + \frac{r+t}{r}F)/t^2}$
6	rtF	n^2m/rt^2	n^2mF/t
8	rtE	n^2m/rt^2	n^2mE/t
11	rtF	n^2m^2/r^2t^2	n^2m^2F/rt
13	rtE	n^2m^2/r^2t^2	n^2m^2E/rt
15	r^2F	n^2m^2/r^2t^2	n^2m^2F/t^2
Operations ($n, m \to \infty$)			$\boldsymbol{n^2m^2X}$
23	X	$t^4n^2m^2/t^4$	n^2m^2X

E. Pseudocode, I/O and Operation Counts of the Register Blocking Primitive

```
1  for I ∈ [0, t, 2t ... n), I' ∈ [0, t, 2t, ... m) do
2      a_{I+[0,t),I'+[0,t)} ← 0
3      for J ∈ [0, r, 2r ... n) do
4          GlobalLoad(A_{I+[0,t),J+[0,r)})                                   LD.G rwF
5          GlobalLoad(E_{I+[0,t),J+[0,r)})                                   LD.G rwE
6          for J' ∈ [0, r, 2r, ... m) do
7              GlobalLoad(A'_{I'+[0,t),J'+[0,r)})                            LD.G rwF
8              GlobalLoad(E'_{I'+[0,t),J'+[0,r)})                            LD.G rwE
9              GlobalLoad(p_{J+[0,r),J'+[0,r]})                              LD.G r²F
10             SharedStore(p_{J+[0,r),J'+[0,r]})                             ST.S r²F
11             parfor i ∈ I + [0, t), i' ∈ I' + [0, t) do
12                 for j ∈ J + [0, r), j' ∈ J' + [0, r) do
13                     SharedLoad(p_{jj'})                                   LD.S F
14                     a_{ii'} ← a_{ii'} + p_{jj'} · A_{ij} · A'_{i'j'}
       · κ_e(E_{ij}, E'_{i'j'})                                              OPS X
15     GlobalStore(a_{I+[0,t),I'+[0,t')})                                    ST.G t²F
```

line	unit cost	iteration count	subtotal
global load $(n, m \to \infty)$			$\boldsymbol{n^2m^2(\frac{w}{r}E + \frac{w+r}{r}F)/t^2}$
4	rwF	n^2m/rt^2	n^2mFw/t^2
5	rwE	n^2m/rt^2	n^2mEw/t^2
7	rwF	n^2m^2/r^2t^2	n^2m^2Fw/rt^2
8	rwE	n^2m^2/r^2t^2	n^2m^2Ew/rt^2
9	r^2F	n^2m^2/r^2t^2	n^2m^2F/t^2
global store $(n, m \to \infty)$			$\boldsymbol{nmF}$
15	F	nm	nmF
shared load $(n, m \to \infty$			$\boldsymbol{n^2m^2F}$
15	F	n^2m^2	n^2m^2F
shared store $(n, m \to \infty)$			$\boldsymbol{n^2m^2F/t^2}$
10	r^2F	n^2m^2/r^2t^2	n^2m^2F/t^2
Operations $(n, m \to \infty)$			$\boldsymbol{n^2m^2X}$
14	X	$t^4n^2m^2/t^4$	n^2m^2X

F. Pseudocode, I/O and Operation Counts of the Tiling & Blocking Primitive

```
1  for I ∈ [0, t, 2t ... n), I' ∈ [0, t, 2t, ... m) do
2      parfor i ∈ I + [0, t), i' ∈ I' + [0, t) do
3          a_{ii'} ← 0
4      for J ∈ [0, t, 2t ... n) do
5          GlobalLoad(A_{I+[0,t),J+[0,t)})                                   LD.G t²F
6          SharedStore(A_{I+[0,t),J+[0,t)})                                  ST.S t²F
7          GlobalLoad(E_{I+[0,t),J+[0,t)})                                   LD.G t²E
8          SharedStore(E_{I+[0,t),J+[0,t)})                                  ST.S t²E
9          for J' ∈ [0, t, 2t ... m) do
10             GlobalLoad(A'_{I'+[0,t),J'+[0,t)})                            LD.G t²F
11             SharedStore(A'_{I'+[0,t),J'+[0,t)})                           ST.S t²F
```

```
12          GlobalLoad(E'_{I'+[0,t),J'+[0,t)})                                   LD.G t²E
13          SharedStore(E'_{I'+[0,t),J'+[0,t)})                                  ST.S t²E
14          GlobalLoad(p_{J+[0,t),J'+[0,t)})                                     LD.G t²F
15          SharedStore(p_{J+[0,t),J'+[0,t)})                                    ST.S t²F
16          parfor i ∈ I + [0,t), i' ∈ I' + [0,t) do
17              for h ∈ [J, J + r, ... J + t) do
18                  SharedLoad(A_{i,h+[0,r)})                                    LD.S rF
19                  SharedLoad(E_{i,h+[0,r)})                                    LD.S rE
20                  for h' ∈ [J', J' + r, ... J' + t) do
21                      SharedLoad(A'_{i',h'+[0,r)})                             LD.S rF
22                      SharedLoad(E'_{i',h'+[0,r)})                             LD.S rE
23                      for j ∈ h + [0,r) do
24                          for j' ∈ h' + [0,r) do
25                              a_{ii'} ← a_{ii'} + p_{jj'} · A_{ij} · A'_{i'j'}
      · κ_e(E_{ij}, E'_{i'j'})                                                     OPS X
26      GlobalStore(a_{I+[0,t),I'+[0,t)})                                        ST.G t²F
```

line	unit cost	iteration count	subtotal
global load $(n, m \to \infty)$			$\boldsymbol{n^2m^2(E+2F)/t^2}$
5	t^2F	n^2m/t^3	n^2mF/t
7	t^2E	n^2m/t^3	n^2mE/t
10	t^2F	n^2m^2/t^4	n^2m^2F/t^2
12	t^2E	n^2m^2/t^4	n^2m^2E/t^2
14	t^2F	n^2m^2/t^4	n^2m^2F/t^2
global store $(n, m \to \infty)$			$\boldsymbol{nmF}$
26	t^2F	nm/t^2	nmF
shared load $(n, m \to \infty)$			$\boldsymbol{n^2m^2(\frac{r+t}{rt}E+\frac{r+t}{rt}F)}$
18	rF	n^2m^2/rt	n^2m^2F/t
19	rE	n^2m^2/rt	n^2m^2E/t
21	rF	n^2m^2/r^2	n^2m^2F/r
22	rE	n^2m^2/r^2	n^2m^2E/r
shared store $(n, m \to \infty)$			$\boldsymbol{n^2m^2(E+2F)/t^2}$
6	t^2F	n^2m/t^3	n^2mF/t
8	t^2E	n^2m/t^3	n^2mE/t
11	t^2F	n^2m^2/t^4	n^2m^2F/t^2
13	t^2E	n^2m^2/t^4	n^2m^2E/t^2
15	t^2F	n^2m^2/t^4	n^2m^2F/t^2
Operations $(n, m \to \infty)$			$\boldsymbol{n^2m^2X}$
25	X	$t^4n^2m^2/t^4$	n^2m^2X