Parallelization of Shortest Path Graph Kernels on Multi-Core CPUs and GPUs

Lifan Xu[1], Wei Wang[1], Marco A. Alvarez[1], John Cavazos[1] and Dongping Zhang[2]

[1] University of Delaware {xulifan, weiwang, malvarez, cavazos}@udel.edu
[2] AMD Dongping.Zhang@amd.com

Abstract. In this paper, we present a study on the parallelization of the shortest path graph kernel from machine learning theory. We first present a fast sequential implementation of the graph kernel which we refer as Fast Computation of Shortest Path Kernel (FCSP). Then we explore two different parallelization schemes on the CPU and four different implementations on the GPU. After analyzing the advantages of each we propose a hybrid version which, for different pairs of graphs, dynamically chooses the best implementation from multicore execution and GPU execution. Finally, we apply our implementations to several datasets that are composed of graphs from different domains. We first evaluate our implementations on a set of synthetic datasets, then, we evaluate our implementations on a set of four real-world graph datasets. The results show that the sequential FCSP algorithm running on CPU is able to achieve a maximum 76x speedup over a naive sequential implementation of the shortest path graph kernel algorithm running on the same CPU. The results also show that our GPU implementation of FCSP offers a maximum 18x speedup over the sequential FCSP. Our GPU implementation also achieves a maximum 2x over a parallel CPU implementation of FCSP.

1 Introduction

Many of the most advanced technologies, e.g. speech recognition in mobile devices, involve rigorous analysis of historical data in the pursuit for patterns and relationships. The discipline of machine learning plays a central role for such data analysis, however, most of their algorithms still require substantial computational resources. Furthermore, in several real world applications, data occur naturally as complex discrete structures demanding even more computational power.

Examples of such structures include: social networks where social actors are connected to each other by interactions [1]; biomolecules, DNA/RNA sequences, proteins, and chemical compounds that are important bodies in the fields of chemoinformatics [2]; and bioinformatics [3]. These data observations can typically be expressed by graphs, which are powerful formalisms that naturally encode the structural relationships present in the data and allow for computational processing.

In fact, one increasingly popular approach for learning patterns from databases of graphs is the design of kernels for graphs. A *kernel function* in machine learning can be roughly understood as a similarity function between a pair of input graphs. Powerful learning algorithms known as kernel methods, such as the Support Vector Machine [4], interact with data observations exclusively through kernel functions and they have been extensively used across several scientific applications [4–7].

An attractive kernel function for graphs is based on counting similar shortest paths in a pair of input graphs [8]. In contrast to most existing kernels for graphs, kernels based on shortest paths can deal with labeled graphs, where the labels are not restricted to a discrete alphabet, that is, one can label nodes and/or edges with continuous values or even more complex compositions. Graph kernels have proven [9] efficient with small databases of graphs, however, because of the inherent complexity of graphs in real world scenarios and the need to scale to larger graphs, graph kernels can have high computational costs.

In this paper, we focus on accelerating graph kernels based on shortest paths, as originally proposed by Borgwardt et al. [8]. Research has shown that these kernels are highly competitive in terms of accuracy and running time, when compared with others [9]. To the best of our knowledge, no other work has addressed the parallelization of shortest path graph kernels. Note that, the sequential version of this algorithm runs in $O(n^4)$, which makes it only appropriate for instances of not very large graphs.

For a given dataset $D = \{g_1, g_2, \ldots, g_n\}$ of graphs, our experiments focus on the calculation of the corresponding kernel matrix $M_{n \times n}$, a symmetric matrix where every element $M(i, j) = SPGK(g_i, g_j)$ refers to the shortest path graph kernel function applied to a pair of input graphs g_i and g_j.

Our proposed method splits the original shortest path kernel into two parts and makes the calculation much faster. We call it the Fast Computation of Shortest Path Kernel, referred as FCSP. We explored two different parallelization schemes of FCSP on the CPU using OpenMP. One focuses on the parallelization of FCSP on a single pair of graphs while the other focuses on the parallelization of calculating the entire kernel matrix. Next, we split the FCSP into three different GPU kernels using OpenCL which will calculate respectively, the pairwise similarities between the vertices of the input graphs, the edges from the two input graphs, and the aggregation of those similarities into the final value for $FCSP$ on the GPU. We implement four different GPU parallelizations. The first uses a 1D scheme for domain decomposition, the second uses a 2D scheme for domain decomposition. The third and fourth overlap communication and computation for the first and the second method. We observed that the OpenMP implementations work better when the input graphs are small, while the GPU implementations are better for larger graphs. This information suggests a hybrid implementation combining CPU parallelization with the best GPU parallelization. The hybrid scheme is based on a graph size threshold. Graphs smaller than the threshold size are assigned the CPU parallelization. Larger graphs are assigned the GPU parallelization.

To measure the performance of our different implementations, we perform two separate experiments, where we apply our different accelerated codes to several datasets. In the first experiment, we create 9 synthetic datasets. Graphs in the same dataset have exactly the same number of nodes, and they are all fully connected. We also created one additional dataset containing graphs with different sizes to test the performance of the hybrid implementation. In the second experiment, we measured the speedups in the kernel matrix calculation for different samples of graphs from real-world scientific datasets; specifically four datasets from the bioinformatics domain are used. As expected, the hybrid implementation achieved the best performance because size of graphs can vary from less than ten nodes to over one hundred nodes in these datasets.

This paper is organized as follows. In Section 2, we present the shortest path graph kernel. In Section 3, we propose a fast sequential computation of the Shortest Path Graph Kernel. In Section 4, we introduce two different CPU parallelizations of the FCSP algorithm. In section 5, we describe our different GPU implementations and the hybrid method. Section 6 presents our experimentation and the analysis of the results. In Section 7, we present an overview of related work. Finally in Section 8, we discuss our conclusions and future work.

2 Shortest Path Graph Kernel

Graph kernels based on shortest paths were proposed by Borgwardt and Kriegel [8]. Roughly speaking, this kernel counts the number of shortest paths of the same length having similar start and end vertex labels in two input graphs. One of the motivations for using this kernel is that it avoids the problem of "tottering" found in graph kernels that use random walks [10]. Tottering is the act of visiting the same nodes multiple times thereby artificially creating high similarities between the input graphs. In shortest path kernels, vertices are not repeated in paths, thus, tottering is avoided.

In practice, a graph kernel based on shortest paths will require to determine all shortest distances in a graph, a problem that is solvable in polynomial time. For example, the Floyd-Warshall algorithm [11] calculates the shortest distances for all pairs of nodes in $O(n^3)$ time, where n denotes the number of vertices. This algorithm allows graphs with negative edge labels, but not containing any negative cycles, which happen when all edge labels in the cycle sum to a negative value. In order to define a kernel that counts shortest paths of similar distances, the original graphs must be transformed into shortest path graphs. This step is a preprocessing requirement before calculating the shortest path graph kernel.

Given a graph $G = \langle V, E \rangle$, a shortest path graph is a graph $G_{sp} = \langle V', E' \rangle$, where $V' = V$ and $E' = \{e'_1, \ldots, e'_m\}$ such that $e'_i = (u'_i, v'_i)$ if the corresponding vertices u_i and v_i are connected by a path in G. The edges in the shortest path graph are labeled with the shortest distance between the two nodes in the original graph.

A shortest path graph kernel for two shortest path graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is defined as:

$$k_{sp}(G, G') = \sum_{e \in E} \sum_{e' \in E'} k_{walk}(e, e') \tag{1}$$

where k_{walk} is a positive definite kernel for comparing two edge walks of length 1.

The edge walk kernel k_{walk} is the product of kernels on nodes and edges along the walk. Since the length of the walk is 1, k_{walk} can be calculated in terms of the start vertex, the end vertex, and the edge connecting

both. Let e be the edge connecting vertices u and v, and e' be the edge connecting nodes u' and v'. The edge walk kernel is defined as follows:

$$k_{walk}(e, e') = k_{node}(u, u') \cdot k_{edge}(e, e') \cdot k_{node}(v, v') \tag{2}$$

where k_{node} is a valid kernel function for comparing two vertices, and k_{edge} is a valid kernel function for comparing two edges. The positive definiteness of the kernel in Eq. 1 follows from its definition as a R-convolution kernel [12]. Pseudo-code for a naive implementation of the Shortest Path Graph Kernel is presented in Alg. 1. Given two input graphs $g1$ and $g2$, line 2-7 loops over the shortest path matrices to find all pairs of paths. Line 8 calculates the k_{edge} and line 10-11 calculates k_{node}. Line 12 calculates k_{walk} and sum it up.

This kernel is attractive because it retains expressivity while avoiding tottering. Moreover, it can be applied to all graphs on which Floyd-Warshall can be performed, as well as the fact that it allows for continuous labels in vertices and edges. The runtime complexity of this kernel is $O(n^4)$, because the Floyd-Warshall transformation can be done in $O(n^3)$ and the kernel calculation requires a pairwise comparison on the number of edges of the shortest path graphs. The latter takes $O(n^2 * n^2)$, because in the worst case the shortest path graph is complete, having n vertices and $\frac{n(n-1)}{2}$ edges.

Algorithm 1 Shortest Path Graph Kernel Algorithm

```
1: K ← 0
2: for i, j = 0 → n_node[g1] do
3:     w1 ← sp_mat[g1][i][j]
4:     if i ≠ j AND w1 ≠ INF then
5:         for m, n = 0 → n_node[g2] do
6:             w2 ← sp_mat[g2][m][n]
7:             if m ≠ n AND w2 ≠ INF then
8:                 k_edge ← EdgeKernel(w1, w2)
9:                 if k_edge > 0 then
10:                    k_node1 ← NodeKernel(g1, g2, i, m)
11:                    k_node2 ← NodeKernel(g1, g2, j, n)
12:                    K+ = k_node1 * k_edge * k_node2
13:                end if
14:            end if
15:        end for
16:    end if
17: end for
18: return K
```

3 Fast Computation of the Shortest Path Graph Kernel

A naive implementation of the Shortest Path Graph Kernel (Alg. 1) has three disadvantages that may slow down its performance. The first is the number of control flow operations. Four for loops and two if statements definitely slow down the program performance whether sequential or parallelized. The second defect is potential redundant computation of k_{node}. Let us consider two graphs as shown in Fig. 1(a). When we compare walk $D- > E$ with $A- > B$, we need to compute k_{node} on (D, A) and (E, B). When we compare walk $D- > F$ with $A- > B$, k_{node} on (D, A) and (F, B) have to be calculated. So there is a redundant calculation of k_{node} on (D, A) which is a waste of computation resource and time. The third drawback is the random memory access pattern in Alg. 1. Sequential memory access is preferred on the CPU, and this is even more true for SIMD architectures like the GPU. The sequential and random read bandwidths on a Nehalem CPU and Fermi GPU have been measured before [13], results as shown in Table 1. As you can see, there is a 9x difference in bandwidth on the CPU, and 28x difference on the GPU.

To address the issues of Alg. 1 we propose a different way to calculate the shortest path graph kernel. We refer to this as the Fast Computation of Shortest Path Graph Kernel ($FCSP$). In our method, the calculation of the shortest path graph kernel is divided into two main components. First, we calculate all possible instances of k_{node} into a vertex kernel matrix. Second, we calculate all the required values for k_{walk}. Note that the kernel

Table 1. Sequential and Random Memory Read Bandwidth on CPU and GPU

Platform	Sequential Read	Random Read
Nehalem CPU	8.6 GB/s	0.9 GB/s
Fermi GPU	76.8 GB/s	2.7 GB/s

functions k_{node} and k_{edge} used to calculate the similarity between a pair of nodes and a pair of edges, can be different from application to application. In our experiments, we use the Gaussian kernel and the Brownian Bridge kernel which are positive semidefinite [14].

For the first component, named $VertexKernel$, we proceed as follows. Assuming that the order of g_1 is m and the order of g_2 is n, we create a matrix $V_{m\times n}$ for storing the k_{node} values, where every entry is the value of $k_{node}(u, u')$ for u being a node of g_1 and u' being a node of g_2. By using this scheme, the redundant computation of k_{node} is eliminated.

The second component, named $WalkKernel$, is responsible for calculating k_{walk}, and takes advantage of a new representation of the shortest path adjacency matrix. The new representation is composed of three equally-sized arrays. The length of these arrays is the number of edges in the corresponding matrix. The three arrays store respectively: the weight of the edge, the index of the starting vertex, and the index of the ending vertex. This representation is inspired by the formats of storing a sparse matrix on GPUs [15] which can solve the low memory utilization problem for sparse matrices access. By applying this transformation, the two if statements in Alg. 1 can be removed and four for loops are reduced to two.

The pseudo-code of our new method is presented in Alg. 2. Given input graphs $g1$ and $g2$, function $Vertex\_Kernel$ calculates all possible instances of k_{node} sequentially and stores them in a matrix V for later access. Function $Walk\_Kernel$ takes advantage of the three 1D arrays converted from shortest path matrix, satisfies more sequential memory access and less branch divergence. It calculates all k_{walk} and sums them up as the final similarity between two input graphs.

Algorithm 2 Fast Computation of Shortest Path graph kernel

```
1: function VERTEX_KERNEL
2:     for i = 0 → n_node[g1] do
3:         for j = 0 → n_node[g2] do
4:             V[i][j] ← NodeKernel(g1, g2, i, j)
5:         end for
6:     end for
7: end function
8:
9: function WALK_KERNEL
10:    K ← 0
11:    for i = 0 → n_node[g1] do
12:        x1 ← edge_x1_g1[i]
13:        y1 ← edge_y1_g1[i]
14:        w1 ← edge_w1_g1[i]
15:        for j = 0 → n_node[g2] do
16:            x2 ← edge_x2_g2[j]
17:            y2 ← edge_y2_g2[j]
18:            w2 ← edge_w2_g2[j]
19:            k_edge ← EdgeKernel(w1, w2)
20:            if k_edge > 0 then
21:                k_node1 ← V[x1][x2]
22:                k_node2 ← V[y1][y2]
23:                K+ = k_node1 * k_edge * k_node2
24:            end if
25:        end for
26:    end for
27:    return K
28: end function
```

4 FCSP on Multi-Core CPU

In our experiments we pursue the calculation of a kernel matrix from a given input dataset of n graphs. A kernel matrix is a symmetric matrix where every entry $M[i, j]$ for $i, j \leq n$ is the corresponding shortest path graph kernel between graphs g_i and g_j. Here we present two different schemes of $FCSP$ parallelization on multicore CPUs. Both schemes are implemented using OpenMP.

In the first scheme referred as $OpenMP\_In$, we make the $FCSP$ computation on a single pair of graphs running in parallel. For the first part of $FCSP$, which is $Vertex\_Kernel$, we create a shared 2D matrix for all the OpenMP threads. Then we parallelize the outside loop, which is line 2 in the $VertexKernel$ as shown in Alg. 2, using the dynamic parallel for pragma. In the $Walk\_Kernel$, we create an array with size equal to the number of OpenMP threads. This array stores the summed k_{walk} from all OpenMP threads. We use the same dynamic parallel for pragma to parallelize line 11 of Alg. 2 because dynamic parallel is observed faster than the static parallel scheme.

The second scheme is parallelization of the kernel matrix calculation which is referred as $OpenMP\_Out$. In this scheme, we create the same number of threads as the number of CPU cores available. Each thread resides on one core. To calculate the symmetric kernel matrix for a set of input graphs, the top half triangle of the matrix is transformed to a 1D array. Each OpenMP thread takes one element in the 1D array in order, applies the $FCSP$, fills in the result, and then goes to the next iteration until all elements are computed.

5 FCSP on GPU

The GPU is a massively parallel co-processor present in desktops and laptops. The most powerful GPUs can perform more FLOPS and have more memory bandwidth than the most powerful CPUs [16]. Moreover, the development of programming environments, like CUDA and OpenCL, allows programmers to run multiple threads in parallel using the power of the GPU for general-purpose applications.

The CUDA and OpenCL models use an SPMD model where a program known as a kernel represents a single scalar execution entity. In CUDA terminology, which we will use throughout for simplicity, this is known as a thread. These CUDA threads are organized for execution into 1D, 2D or 3D grid of structures known as thread blocks which perform local computation and which are co-located on the same execution unit of the GPU. Thread blocks have access to a shared on-chip shared memory and can synchronize their constituent threads with each other. Threads are further executed in 16-, 32- or 64- element batches called warps where each thread is a single SIMD lane and a warp is thus analogous to a x86 thread executing an SSE or AVX instruction. This mapping of neighboring threads to a single SIMD vector leads to efficiency losses, known as thread divergences, when threads are mapped to the same warp follow control flow paths.

We present our different GPU parallelizations of the $FCSP$. We first introduce two different domain decomposition techniques for $FCSP$ parallelization. Then we reduce the total running time of these two implementations by overlapping communication and computation. After that, we propose a hybrid method that combines multicore CPU and GPU parallelization.

5.1 Two Domain Decompositions in GPU Parallelization

$FCSP$ is naturally applicable for parallelization. In this implementation, branches are removed, no load balancing issue exists between GPU threads, and the coalesced memory access is satisfied. We are therefore able to achieve significant speedups with this approach.

In our GPU implementation, the $FCSP$ is divided into three GPU kernels. The first one is $Vertex\_Kernel$. It calculates all possible instances of k_{node} and stores them in a matrix for later access. The second kernel is $Walk\_Kernel$ which calculates all the required values for k_{walk} and stores them in a matrix or array. The last component is $Reduction\_Kernel$ which sums up all k_{walk} values into a small array. The small array is copied to CPU memory and summed up as the final similarity.

For the first component, named $Vertex\_Kernel$, we proceed as follows. Assuming that g_1 has m vertices and g_2 contains n, we allocate a buffer $V_{m \times n}$ on the GPU memory for storing the k_{node} value. A GPU thread grid is created, where each thread calculates an entry of V. As we remove the divergence, all threads in this component are running in parallel.

The second component, named $Walk\_Kernel$, is responsible for calculating k_{walk}. Given two input graphs, suppose the number of paths in g_1 is a and g_2 has b paths. We assume g_1 has more paths than g_2 without

loss of generality. For the graph with n nodes, the paths can vary from 0 to n^2. So the domain decomposition for GPU threads can be tricky. In our implementation, we tried two different methods. The first method is 1D decomposition in which we assign a GPU thread to one path in g_1. This thread will loop through all the b paths in g_2, calculate the corresponding k_{walk} value and sum them up. An array of a elements will be returned at the end. The second scheme is 2D decomposition. In this method we assign one GPU thread to one pair of paths. So each thread will calculate k_{walk} between two paths. A matrix of $a \times b$ will be returned. The calculation of k_{walk} requires k_{node}, that has already been calculated and cached.

The third GPU kernel is $Reduction\_Kernel$. If we used the 1D domain decomposition scheme in $WalkKernel$, a reduction is performed on a elements. Otherwise, the reduction is performed on $a \times b$ elements. After reduction, a small result array is copied back to the CPU. Finally, the similarity between the graphs is calculated by adding up all the values in the array.

The biggest advantage of parallelizing $FCSP$ on GPU is efficiency. There is no execution divergence between threads thanks to the shortest path matrix conversion in $Vertex\_Kernel$ and $Walk\_Kernel$. The sequential coalesced memory access is satisfied in all three kernels.

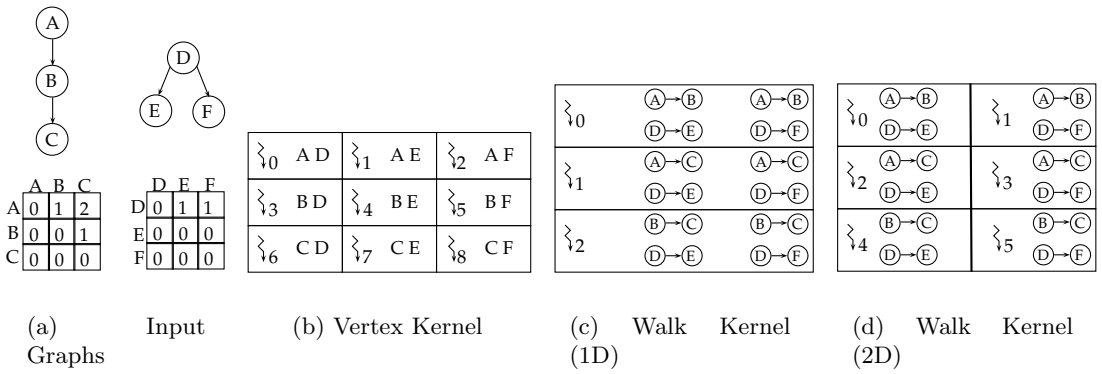


Fig. 1. Example for applying Shortest Path Graph Kernel using *Advanced*. Figure 1(a) shows the input graphs and the corresponding shortest path adjacent matrix. Figure 1(b) depicts the $Vertex\_Kernel$ and each GPU thread's assignment. Figure 1(c) shows the $Walk\_Kernel$ with 1D domain decomposition and each GPU thread's calculations. Figure 1(d) shows the $Walk\_Kernel$ with 2D domain decomposition and each GPU thread's calculations.

To make it easy to understand, we show a simple example in Figure 1. Figure 1(a) shows the two input graphs and the shortest path adjacency matrix. Figure 1(b) demonstrates how $Vertex\_Kernel$ calculates $k_{node}(u, u')$ for the inputs. Since there are three vertices in each graph, we create a thread grid of size 3×3, as shown in the figure. Each thread in the grid is responsible for calculating one $k_{node}(u, u')$. Results are stored in a matrix and can be cached for later access. Figure 1(c) shows the $Walk\_Kernel$ with 1D decomposition. Since there are three edges in the first input graph, we create three GPU threads. Each thread loops through the two edges in the other input graph. The k_{node} values for its vertices are pre-calculated and cached. This allows the threads to finish the calculation of k_{walk} extremely fast. Figure 1(d) shows the $Walk\_Kernel$ with 2D decomposition. Since there are three paths in one graph and two paths in the other one, six GPU threads are created. Each thread calculates the k_{walk} between one pair of paths.

Before the GPU kernel execution, the two input graphs have to be copied to the GPU memory. So if there are n comparisons, n memory transfers between the CPU and the GPU are needed. This will result in a huge overhead. To avoid the unnecessary and duplicated memory transfers, we can simply copy all the graph data into GPU memory. Then at each kernel execution, the GPU thread can fetch needed data according to its targeting graph offset.

5.2 Overlapping Communication with Computation

In our GPU implementation, the last kernel is the $Reduction\_Kernel$. A small array is copied to the CPU and summed up for calculating the final similarity. This memory copy from GPU to CPU and computation on CPU

may not take too much time. However, given n input graphs, the GPU method needs to be called n^2 times. As a result, there may be considerable time spent on memory transfers and CPU computation. Our experiments show that the portion of total time spent on the reduction memory transfer can vary from 6% to 50%. Fortunately, this part can be hidden by overlapping it with GPU computation. Here is how it works. When the reduction kernel completes, we initiate a non-blocking memory transfer then assign another pair of graphs to the $Vertex\_Kernel$. As the memory transfer is asynchronous it can be overlapped with the following $Vertex\_Kernel$ execution. When $Vertex\_Kernel$ completes we initiate a non-blocking execution of $Walk\_Kernel$ and the CPU accumulates the result array to obtain the similarity result while the GPU is executing. Our experiments show this scheme can hide most of the time spending on memory transfer other than the function call overhead.

5.3 Hybrid Implementation – Combining CPU and GPU

From our experiments, we observed that the implementation with the best performance may be different in different datasets. When the graphs are really small, the CPU implementation beats all the other GPU implementations. The GPU with 2D decomposition beats the GPU with 1D decomposition when graphs are small, but it cannot beat the OpenMP implementations. However, when the graphs get bigger, the GPU with 1D decomposition performs the best. The experiments show that the overlapped implementations always perform better than the ones without overlapping. So we think it should be a good idea to combine CPU and GPU implementation together. We hypothesize that many real world datasets have graphs of a variety of different sizes. So in our *Hyrbrid* implementation, we set one threshold T about graph sizes. When the number of shortest paths in both input graphs are smaller than T, we use $OpenMP\_In$ to calculate the similarity. Otherwise, the $GPU\_1D\_overlap$ is used.

6 Experiments

All the experiments were conducted on a GPU cluster where each node has a NVIDIA C2050 GPU. This GPU is based on the GF100 (Fermi) architecture. It contains 14 multiprocessors with 32 processors each, for a total of 448 parallel processors. Each multiprocessor contains 32K registers and 64KB, which are split between shared memory and L1 cache. Programmers can allocate 16KB for shared memory and 48KB for L1 cache or 48KB for shared memory and 16KB for L1 cache. In addition to the GPUs, each node contains two Intel 5530 Quad core Nehalem CPUs clocked at 2.4 GHz with 8MB cache. For our OpenMP implementation, we used sixteen CPU threads.

We tested our GPU accelerated versions of the shortest path graph kernel using two datasets. The first dataset is synthetic. The second dataset is a scientific dataset containing labeled graphs using discrete values. For performance comparisons, we do take memory copies and all the other overhead into consideration, i.e., the total running time is used.

6.1 Results on Synthetic Datasets

To test the performance of all our implementations, we created some synthetic datasets. Our OpenMP implementations perform well on datasets with small graphs, while the GPU implementations perform better on datasets with larger graphs.

Dataset First, we created nine different datasets. We call these *homogeneous* datasets because each dataset contains graphs of same sizes. Graphs in the same set have the same number of nodes. All the graphs are fully connected which means for each pair of vertices, there is an edge connecting them. Since the graphs are fully connected, the number of Shortest Paths (SP) equals to the number of edges. Each dataset has 256 graphs. The 9 datasets contain graphs with 10, 15, 20, 25, 30, 35, 40, 45, and 50 nodes. The largest number of nodes we use is 50, this is because the average number of nodes in the real scientific datasets we are going to test is less than 50. However, we are able to process large graph with thousands of nodes, as long as it can fit into GPU memory. If the graph size goes beyond the GPU memory capability, we can still cut a graph into multiple chunks and process them chunk by chunk. However, in this paper we do not present results on very large graphs.

We first evaluated the naive sequential implementation of the shortest path graph kernel and the $FCSP$ on the CPU. Then we evaluated our two different OpenMP implementations and our four different GPU implementations on the synthetic datasets. Table 2 shows statistics for all 9 datasets. In order to test the performance

of our *Hybrid* implementation, we create another dataset with mixed sized graphs. In the mixed dataset, we create 180 10-nodes graphs and 76 50-nodes graphs. Hence the comparison between 10-nodes graphs can be handled by *OpenMP_In*, the comparison between 10-nodes graphs and 50-nodes graphs, and the comparison between 50-nodes graphs can be handled by *GPU_1D_overlap*. We pick the 180:50 ratio because the number of graphs assigned to CPU would roughly equal to the number of graphs assigned to GPU in this case.

Table 2. Statistics about the number of nodes and edges for synthetic datasets.

Dataset	Avg. Nodes	Avg. Edges	Avg. SP
10-nodes	10	90	90
15-nodes	15	210	210
20-nodes	20	380	380
25-nodes	25	600	600
30-nodes	30	870	870
35-nodes	35	1190	1190
40-nodes	40	1560	1560
45-nodes	45	1980	1980
50-nodes	50	2450	2450

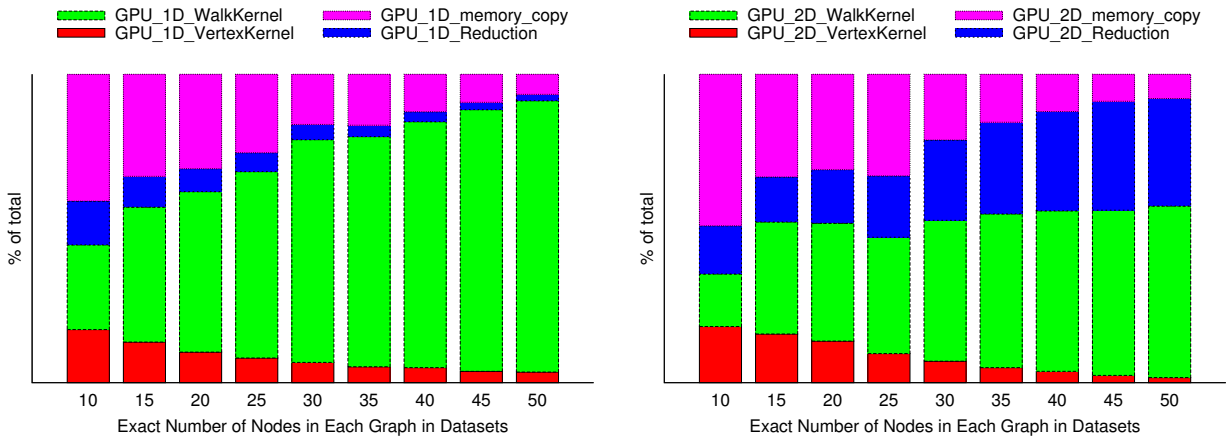
Results Table 3 shows the total running time in seconds of the naive implementation and the FCSP on 9 synthetic datasets. Thanks to the branch divergence removal, redundant computation elimination, and sequential memory access, our sequential FCSP algorithm running on a CPU is able to achieve a 76X speedup over the naive sequential SPGK algorithm running on the same CPU.

Table 3. Speedup of FCSP over naive SPGK on CPU

Dataset	SPGK time(sec)	FCSP time(sec)	Speedup
10-nodes	127.99	2.35	54.56
15-nodes	695.24	11.69	59.46
20-nodes	2275.60	37.16	61.25
25-nodes	5668.74	91.42	62.00
30-nodes	11990.24	190.82	62.83
35-nodes	26220.74	355.50	73.76
40-nodes	45850.24	609.67	75.21
45-nodes	74817.26	983.35	76.08
50-nodes	115728.37	1513.69	76.45

After comparing our FCSP with the naive implementation, we assess six different FCSP parallelizations on CPU and GPU. For *GPU_1D* and *GPU_2D*, we measured the running times spent on the three GPU kernels and reduction memory copy. The time breakdown is showing in Fig. 2. As the graph size increases the ratio of $WalkKernel$ is getting bigger and bigger, the percentages for $VertexKernel$ and memory copy decreases in both *GPU_1D* and *GPU_2D*. The percentage for *Reduction* goes up in *GPU_2D*, this is because the total number of k_{walk} values to be summed increases exponentially (n^2) as the graph size increases. However, in *GPU_1D*, the increase is linear. The percentage for memory copy vary from 6% to 50% of the total running time in different datasets. So it is necessary to hide this cost.

The speedup of all parallelizations over sequential FCSP are shown in Figure 3. The x-axis shows the exact number of nodes in each graph in the corresponding dataset while the y-axis shows the speedup. As you can see, *OpenMP_Out*'s performance is stable reaching almost 8X speedup on average. This is reasonable because there are sixteen OpenMP threads running in parallel in a shared memory system. Even FCSP is optimized, it is still memory bandwidth bounded which prevents it from 16x speedup. In *OpenMP_In* method, the overhead for OpenMP initialization occurs once for one pair of graphs. So the initialization happens $\frac{n(n-1)}{2}$ times given n input graphs. This is the main reason why *OpenMP_In* performs worse than *OpenMP_Out* especially when the graph size is small. For the GPU implementations, the overlapped implementations are fast than the non-overlapped



(a) Time breakdown for GPU_1D

(b) Time breakdown for GPU_2D

Fig. 2. Time breakdown for the GPU_1D and GPU_2D implementation on the nine datasets. (a) shows the running times in percentages for the $VertexKernel$, $WalkKernel$, $Reduction$, and memory copy for GPU_1D on nine synthetic datasets, and (b) shows the running times in percentages for the GPU_2D.

implementations. It proves that hiding memory copy cost by overlapping communication with computation can help reducing total running time. The $GPU\_1D\_overlap$ performs best in four GPU parallelization methods on almost all datasets except the first one. Also, $GPU\_1D\_overlap$ starts to outperform OpenMP implementations when the graph sizes increases to 35. It reaches a speedup of 18X on the largest dataset.

Table 4. Different Implementation Running Time(seconds) on the Mixed Dataset

$OpenMP\_In$	$OpenMP\_Out$	GPU_1D	GPU_2D	$GPU\_1D\_overlap$	$GPU\_2D\_overlap$	$Hybrid$
24.446	20.349	22.137	32.252	19.751	29.336	**19.042**

To test the performance of our $Hybrid$ implementation, one additional dataset was created. In this dataset, only two sizes of graphs were included: five node graphs and fifty node graphs. Thus, there is a clear boundary we can set in our $Hybrid$ implementation for choosing which algorithm to use. Our $Hybrid$ algorithm uses $OpenMP\_In$ when graph similarities of five nodes are calculated. It then switches to the $GPU\_1D\_overlap$ implementation when similarities of large graphs are calculated. Table 4 shows the running time in seconds of our different implementations on the mixed dataset. As can be seen, $Hybrid$ performs the best which fits our expectation.

6.2 Results on Scientific Datasets

Datasets We also carried out an experiment with real-world scientific datasets from the bioinformatics domain, with graphs that contain discrete labels at the nodes. These datasets were used in prior work to highlight the effectiveness and efficiency of shortest path graph kernels [9]. The datasets are described as follows: (a) MUTAG contains mutagenic aromatic and heteroaromatic nitro compounds [17]; (b) ENZYMES is a dataset of protein tertiary structures of enzymes from the Brenda database [18]; (c) NCI1 and NCI109 are two datasets of chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines, respectively [19]. Detailed statistics about these datasets are shown in Table 5.

Results In this experiment, we used each of the four scientific datasets as input to our two OpenMP implementations, four GPU implementations, and one hybrid implementation. We show the time breakdown for GPU_1D on four scientific datasets in Fig. 4. It shows that the time spent on memory copy takes up to 31% in MUTAG

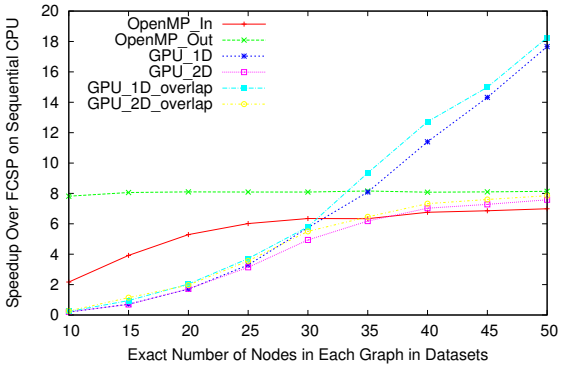


Fig. 3. Speedup over sequential FCSP on 9 synthetic homogeneous datasets

Table 5. Detailed statistics about the number of nodes and edges for the four scientific datasets.

Dataset	Num. of Graphs	Avg. Nodes	Avg. Edges	Min. SP	Max. SP	Avg. SP
MUTAG	188	17	39	90	756	324
ENZYMES	600	32	124	2	15500	1215
NCI1	4110	29	64	6	11130	1005
NCI109	4127	29	64	12	11130	995

which has the smallest average number of SP, and 17% in ENZYMES which has the largest average number of SP. We use the performance of *OpenMP_In* as a baseline for comparison. The results from all the other implementations are shown in Table 6. As you can notice, the overlapped GPU implementations outperform the non-overlapped methods in all four datasets. In the first dataset MUTAG, the *OpenMP_Out* performs the best. This is because the dataset is really small. It only has 324 shortest paths in average. The computation power of GPU cannot be fully utilized in this dataset. As the result, all GPU implementations including *Hybrid* perform not so well. In the other three datasets, *Hybrid* always performs the best. We also notice that *Hybrid* achieves its maximum speedup for ENZYMES, which is the dataset with the largest average number of nodes and edges.

Table 6. Speedup over *OpenMP_In* on four scientific datasets

Dataset	*OpenMP_Out*	*GPU_1D*	*GPU_2D*	*GPU_1D_overlap*	*GPU_2D_overlap*	*Hybrid*
MUTAG	1.33	0.28	0.21	0.33	0.38	0.96
ENZYMES	1.05	1.73	0.78	1.89	0.94	1.92
NCI1	1.13	1.45	0.73	1.66	0.90	1.78
NCI109	1.11	1.38	0.69	1.58	0.87	1.70

7 Related Work

While the running time of the shortest path graph kernel is an obvious improvement over other graph kernels, it is still expensive for large graphs or datasets. No other related work presents a GPU implementation of the shortest path graph kernel. However, the transformation of a graph into a shortest path graph on the GPU, in particular the Floyd-Warshall algorithm, has been done in the past. Harish and Narayanan [20] presented a simple GPU implementation. They assign each atomic task to a single GPU thread. Their approach is limited by the time spent on accessing global memory. Katz and Kider [21] improved Harish's work by using a blocked approach. In their implementation, shared memory is used, which resulted in a 5x speedup over Harish's work. Lund and Smith [22] applied a multi-stage approach, where they can remove data dependencies and make a more efficient use of registers and shared memory. In the end, they achieved a 5x speedup over Katz's implementation.

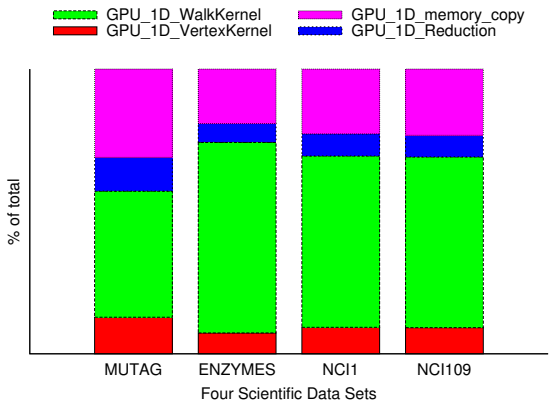


Fig. 4. Time breakdown for GPU_1D on four scientific datasets

In all our GPU implementations, the application of the Floyd-Warshall algorithm is done on the CPU when the graphs are read as input. This is because the Floyd-Warshall algorithm only takes a small amount of time which is less than 1% of the total running time in all our experiments. Certainly, our approaches could be improved by using a GPU implementation for the Floyd-Warshall algorithm [20] [21] [22]. However, in the context of this paper, we focus on accelerating only the shortest path graph kernel, so this is out of the scope of the current paper.

8 Conclusion

In this paper, we are targeting fast and efficient parallelization of the shortest path graph kernel on the CPU and GPU. We proposed the Fast Computation of Shortest Path graph kernel which is able to achieve 76x speedup over a naive implementation of the SPGK if we run both of them sequentially. We parallelized FCSP on the CPU using two OpenMP methods, and four OpenCL implementations on the GPU. We also come up with a hybrid scheme to combine the advantage of CPU and GPU parallelization. Our experiments show that the best implementation of FCSP is dependent on the size of the graphs being processed. For small graphs, the OpenMP implementations on the CPU perform better. GPU implementations perform better if the graphs are large. Therefore a hybrid algorithm that chooses the best algorithm per graph size works best in almost all data sets.

In the future, we plan to extend this work by implementing a multi-GPU version of our algorithm. This will allow us to process larger sets of graphs faster. We will also look into accelerating other graph kernels and handling large graphs that cannot currently fit on the GPU memory.

9 Acknowledgments

This work was funded in part by the U.S. National Science Foundation through the NSF Career award 0953667 and the Defense Advanced Research Projects Agency through the DARPA Computer Science Study Group (CSSG).

References

1. T. Snijders, "Statistical models for social networks," *Annual Review of Sociology*, vol. 37, no. 1, pp. 131–153, 2011.
2. H. Kashima, H. Saigo, M. Hattori, and K. Tsuda, "Graph kernels for chemoinformatics," in *Chemoinformatics and Advanced Machine Learning Perspectives*. IGI Publishing, 2011, ch. I.
3. P. Larrañaga, B. Calvo, R. Santana, C. Bielza, J. Galdiano, I. Inza, J. Lozano, R. Armañanzas, G. Santafé, A. Pérez, and V. Robles, "Machine learning in bioinformatics," *Briefings in Bioinformatics*, vol. 7, no. 1, pp. 86–112, 2006.
4. A. Ben-Hur, C. Ong, S. Sonnenburg, B. Schölkopf, and G. Rätsch, "Support vector machines and kernels for computational biology," *PLoS Computational Biology*, vol. 4, no. 10, pp. e1 000 173+, 2008.
5. J. P. Vert, K. Tsuda, and B. Schölkopf, "A primer on kernel methods," in *Kernel Methods in Computational Biology*, 2004, pp. 35–70.

6. J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004.
7. C. H. Lampert, "Kernel methods in computer vision," *Found. Trends. Comput. Graph. Vis.*, vol. 4, no. 3, pp. 193–285, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1561/0600000027
8. K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," in *IEEE International Conference on Data Mining (ICDM)*, 2005, pp. 74–81.
9. N. Shervashidze and K. Borgwardt, "Fast subtree kernels on graphs," in *Neural Information Processing Systems Conference (NIPS)*, 2009, pp. 1660–1668.
10. P. Mahé, N. Ueda, T. Akutsu, J. Perret, and J. Vert, "Extensions of marginalized graph kernels," in *International Conference on Machine Learning (ICML)*, 2004, pp. 552–559.
11. R. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, pp. 345+, 1962.
12. D. Haussler, "Convolution kernels on discrete structures," UC Santa Cruz, Tech. Rep. UCSC-CRL-99-10, 1999.
13. S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 78–88.
14. B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.
15. N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.
16. NVIDIA, *Nvidia cuda programming guide: Version 3.2*, NVIDIA Corporation, 2010.
17. A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *Journal of Medicinal Chemistry*, vol. 34, no. 2, pp. 786–797, 1991. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/jm00106a046
18. I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg, "Brenda, the enzyme database: updates and major new developments," *Nucleic Acids Research*, vol. 32, no. suppl 1, pp. D431–D433, 2004. [Online]. Available: http://nar.oxfordjournals.org/content/32/suppl_1/D431.abstract
19. N. Wale and G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," in *Proceedings of the Sixth International Conference on Data Mining*, ser. ICDM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 678–689. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2006.39
20. P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th international conference on High performance computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=1782174.1782200
21. G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the gpu," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413957.1413966
22. B. D. Lund and J. W. Smith, "A multi-stage cuda kernel for floyd-warshall," *CoRR*, vol. abs/1001.4108, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1001.html#abs-1001-4108