Designing a Heuristic Cross-Architecture Combination for Breadth-First Search

Yang You*, David A. Bader†, Maryam Mehri Dehnavi‡

*Department of Computer Science and Technology, Tsinghua University, Beijing, China
†School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA
‡Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Boston, MA, USA

Abstract—Breadth-First Search (BFS) is widely used in real-world applications including computational biology, social networks, and electronic design automation. The most effective BFS approach has been shown to be a combination of top-down and bottom-up approaches. Such hybrid techniques need to identify a switching point which is conventionally found through expensive trial-and-error and exhaustive search routines. We present an adaptive method based on regression analysis that enables dynamic switching at runtime with little overhead. We improve the performance of our method by exploiting popular heterogeneous platforms and efficiently design the approach for a given architecture. An 155x speedup is achieved over the standard top-down approach on GPUs. Our approach is the first to combine top-down and bottom-up across different architectures. Unlike combination on a single architecture, a mistuned switching point may significantly decrease the performance of cross-architecture combination. Our adaptive method can predict the switching point with high accuracy, leading to an 695x speedup compared the worst switching point.

Keywords-Graph Algorithm; Data-intensive; Cross-architecture Optimization; Knights Corner MIC; Kepler K20x GPU; Combination; Regression Analysis

I. Introduction

Breadth-First Search (BFS) is widely used in real-world applications including social networks [1], protein interaction analysis [2] and electronic design automation [3]. Top-down and bottom-up are two versions of BFS. Since both top-down and bottom-up have unique advantages over each other, Beamer et al. [4] proposed a combination approach that can switch between top-down and bottom-up in different situations. However, previous naive combinations use trial-and-error and exhaustive search to find the best switching point, which can not be used at runtime because they will significantly increase the execution time. To solve this problem, we design a novel adaptive method based on regression analysis [5]. Compared to the previous naive combination methods, our on-line approach can find the switching point at runtime with little overhead (less than 0.1% of the execution time).

Heterogeneous platforms are becoming more and more popular in recent years because the computing power of co-processors (e.g. GPUs and Xeon Phi) are much stronger than CPUs. For example, each node of the Tianhe-2 supercomputer, which is ranked first on the 41st and 42st Top500 lists [6], contains three Intel Xeon Phi co-processors. To make full use of the existing heterogeneous platforms and improve the performance of the combination method, we propose an effective technique to merge CPU and GPU using the most suitable approach for a given architecture. The proposed approach achieves $8.5\times$, $2.6\times$, and $2.2\times$ average speedup over a MIC combination, a CPU combination, and a GPU combination respectively. Our approach is the first to combine the top-down and bottom-up methods across different architectures. Unlike combination on a single architecture, a mistuned switching point may significantly decrease the performance of cross-architecture combination. Our adaptive method can predict the switching point with high accuracy, leading to $695\times$ speedup compared to the worst switching point. Our contributions are:

(1) an original adaptive method based on regression analysis, which allows the combination technique to be used at runtime and achieves $695\times$ speedup compared to the worst switching point.

(2) the first cross-architecture combination for top-down and bottom-up, which achieves $8.5\times$, $2.6\times$, and $2.2\times$ average speedup over MIC, CPU and GPU combinations respectively.

(3) a pairwise comparison between CPU, GPU and MIC, which can hopefully help the readers select the best architectures for similar applications.

We achieve $16-63\times$ (average $29\times$) speedups over using the Graph 500 benchmark. We also achieve $13\times$ speedup over the state-of-the-art implementation on MIC (Intel Xeon Phi Co-processor).

II. Background

A. Two BFS approaches: top-down and bottom-up

We use $G(V, E)$ to denote a graph where V is the set of vertices and E is the set of edges. Given a vertex v_s, BFS systematically visits every vertex that is reachable from v_s. For a vertex v that is reachable from v_s, v is in level n if the distance from v_s to v is n. The vertices in level $n+1$ will not be visited until all the vertices in level n have been visited. If BFS reaches vertex v via the edges (u, v) from vertex u, we call u the parent or predecessor of v. The general output of BFS is a predecessor map and a level map, which record the parent and level of each vertex.

The pseudocode of the top-down BFS is shown in Algorithm 1. The top-down first finishes the initialization (lines

1-4): putting the source vertex v_s in the current queue (CQ), setting the predecessor of source vertex as itself, and setting the predecessors of all the other vertices as $NULL$. We use the predecessor map ($Pred$) to decide whether a given vertex has been visited (line 9). The top-down then traverses graph until the CQ is empty (lines 5-13). In each level of graph traversal, top-down first empties the next queue (line 6), then visits all the vertices in the CQ (line 7). For a given vertex u in the CQ, the top-down checks all the neighboring vertices of u (line 8). If a neighboring vertex v has not been visited (line 9), it will be added to the next queue (line 10) and its predecessor will be set as u (line 11). After each level of graph traversal, the CQ will be updated by the next queue (line 13).

Another BFS design is the bottom-up approach [4], described in Algorithm 2. The major difference between these two methods is that each vertex in the CQ tries to set all its unvisited neighboring vertices as its children in the top-down approach (line 7-12 in Algorithm 1) while each unvisited vertex searches for one vertex from the CQ as its parent in the bottom-up approach (line 7-12 in Algorithm 2). The top-down approach will always visit $|E|_{cq}$ (the number of edges in CQ) edges while the bottom-up approach at most visits $|E|_{un}$ (the number of edges that have not been visited) edges.

Algorithm 1: top-down approach for BFS

Input: V is the set of vertices;
$\quad\quad\quad E$ is the set of edges;
$\quad\quad\quad v_s$ is the source vertex;
$\quad\quad\quad CQ$ is the current queue for vertices;
$\quad\quad\quad NQ$ is the next queue for vertices.
Output: $Pred$ is the predecessor map.

1   $CQ \leftarrow v_s$
2   **for** $v_i \in V$ **do**
3     $Pred[v_i] \leftarrow -1$
4   $Pred[v_s] \leftarrow v_s$
5   **while** $CQ \neq \emptyset$ **do**
6     $NQ \leftarrow \emptyset$
7   **for** $u \in CQ$ **do**
8   **for** $v \in V$ *and* $(u,v) \in E$ **do**
9   **if** $Pred[v] = -1$ **then**
10          $NQ \leftarrow NQ \cup v$
11          $Pred[v] \leftarrow u$
12   **continue**
13     $CQ \leftarrow NQ$

B. Combination of Top-down and Bottom-up

For most real-world graphs [4], the number of vertices and edges in the CQ are often small at first, then increase and

---

Algorithm 2: bottom-up approach for BFS

Input: the same with Algorithm 1
Output: the same with Algorithm 1

1   $CQ \leftarrow v_s$
2   **for** $v_i \in V$ **do**
3     $Pred[v_i] \leftarrow -1$
4   $Pred[v_s] \leftarrow v_s$
5   **while** $CQ \neq \emptyset$ **do**
6     $NQ \leftarrow \emptyset$
7   **for** $v \in V$ **do**
8   **if** $Pred[v] = -1$ **then**
9   **for** $u \in CQ$ *and* $(v,u) \in E$ **do**
10          $NQ \leftarrow NQ \cup v$
11          $Pred[v] \leftarrow u$
12   **break**
13     $CQ \leftarrow NQ$

---

peak in the middle, and finally become small again (Fig. 1 and Fig. 2). The large number of vertices in CQ are a better candidate for the bottom-up approach because each unvisited vertex will terminate the traversal once its parent is found. With more vertices in CQ, the unvisited vertex can find its parent easier. On the contrary, an increasing number of vertices in the CQ has a negative effect on the top-down approach since the number of edges to travel ($|E|_{cq}$) is increasing.

Fig. 3 shows that bottom-up is much slower than top-down at first. This is because bottom-up has to travel a large number of unvisited edges while the top-down only needs to visit a small number of edges in the CQ. As the level increases, the number of vertices in CQ become larger and peak in the middle. Thus, bottom-up becomes faster than top-down. In the final levels, top-down is slightly better than bottom-up because the number of vertices in CQ decreases significantly compared to the middle part. To improve performance, Beamer et al. [4] proposed a combination technique that can switch between top-down and bottom-up. To show the switching point between top-down and bottom-up, we define two parameters, i.e., M and N. When the number of edges in CQ (i.e. $|E|cq$) is less than $|E|/M$ and the number of vertices in CQ (i.e. $|V|cq$) is less than $|V|/N$, BFS switches to top-down. Otherwise, it switches to bottom-up (Fig. 4).

C. Regression Analysis

Regression analysis [5] is a statistical technique used to model the relationship between a scalar target variable y and a vector sample X. A regression model is first generated based on the training data. The training data contains two parts: $X_i, i \in 1, 2, ..., n$ and $y_i, i \in 1, 2, ..., n$. X_i is a training sample (vector) that contains many features. y_i is
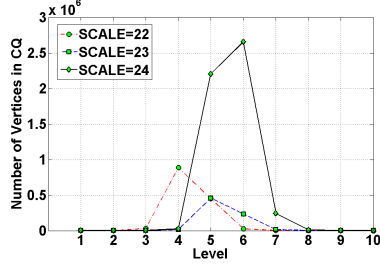
Figure 1. The number of vertices in CQ is small at first, then increases and peaks in the middle. For each graph, the number of vertices is 2^{SCALE}, the number of edges is $2^{SCALE+4}$.
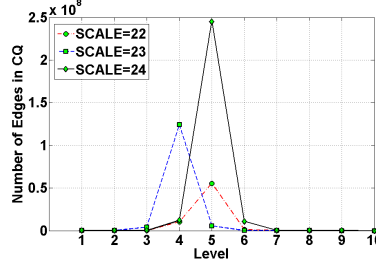

Figure 2. The number of edges in CQ is small at first, then increases and peaks in the middle. For each graph, the number of vertices is 2^{SCALE}, the number of edges is $2^{SCALE+4}$.
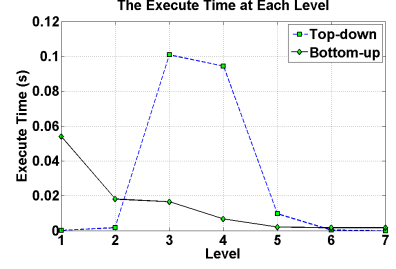

Figure 3. In the beginning bottom-up takes more time than top-down. In the middle bottom-up is faster than top-down. Finally bottom-up becomes slower than top-down.
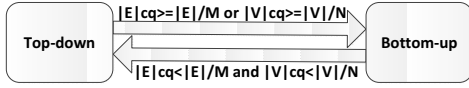

Figure 4. This is the illustration of switching point. When the number of edges in CQ ($|E_{cq}|$) is sufficiently large ($\geq |E|/M$) or the number of vertices in CQ ($|V_{cq}|$) is sufficiently large ($\geq |V|/N$), the program switches to bottom-up. Otherwise, it switches to top-down.

Figure 5. This figure illustrates a simple case of regression analysis. Suppose the training vector X only has one feature, i.e, X can be seen as a scalar. The red nodes are training samples, which we use to generate a model. The model is represented by the line in the figure. Once a new X (e.g. $X = 3$) is obtained, we can predict its target value using the model. The blue triangle is an example of prediction, and the black square is the true value. In practice, there is a difference between the predicted and the true values. A well-trained regression model can minimize this difference.

the target value that corresponds to one and only one training sample X_i. n denotes the number of the training samples (or the target values). A regression model is the relationship between y and X. For example, e.g., a function whose input is a sample and output is a target value. In practice, the target value of a new sample is often unknown. Thus, the regression model can be used to predict the target value based on the information of a new sample. Figure 5 is a simple example of regression analysis.

In this paper, we use Support Vector Machine (SVM) [7] regression. The reason we select SVM over other regression approaches is that SVM is a good candidate for parallel processing on Multi-Core and Many-Core architectures [8], [9]. SVM can also get good prediction accuracy even with small number of training samples [7]. A practical open-source SVM and a detailed tutorial can be found in [10].

D. Related Terms and Parameters

We use the Graph 500 benchmark [11] to describe the graph information and performance metric, the related terms are in Table I. Our experiments are based on the popular Multi-Core (8-core Intel Sandy Bridge CPU) and Many-Core (61-core Intel Knights Corner MIC and 2496-core NVIDIA Kepler K20x GPUs) architectures. The related parameters of these architectures are listed in Table II.

III. ADAPTIVE COMBINATION

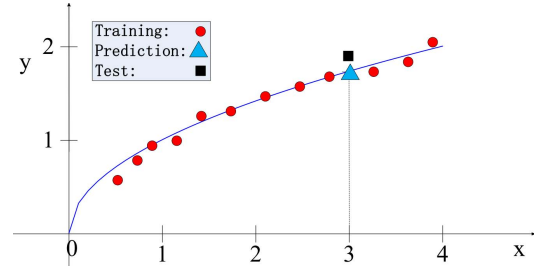We illustrate the adaptive combination technique in this section, and then present the cross-architecture optimization

Table I
RELATED TERMS USED IN GRAPH 500 [11]

Terms	Descriptions
$TEPS$	Traversed Edges Per Second, the performance metric of BFS
$SCALE$	The logarithm base two of the number of vertices
2^{SCALE}	The number of vertices
$edgefactor$	Half the average degree of a vertex in the graph
$2^{SCALE} \times edgefactor$	The number of edges
A, B, C, D	Statistical parameters used in graph construction (Section V)

in Section IV. In order to obtain the best switching point between top-down and bottom-up, we need to get the best settings for M and N. We will only illustrate how to get the best M. The best N can be obtained the same way.

A. Algorithm and Parallelism Comparison

The computational complexity of the conventional top-down approach is $\Theta(V + E)$ [12] and for a sparse graph

Table II
ARCHITECTURE PARAMETERS

Architecture	CPU	MIC	GPU
Frequency (GHz)	2.00	1.09	0.73
DP Peak Performance (Gflops)	128	1010	1320
SP Peak Performance (Gflops)	256	2020	3950
L1 cache (KB)	32/core	32/core	64/SM
L2 cache (KB)	256/core	512/core	1536/card
L3 cache (MB)	20/socket	0	0
Coherent cache	L3	L2	L2
Theoretical bandwidth (GB/s)	51.2	352	250
Measured bandwidth (GB/s)	34	159	188
SP RCMB (flops/B)	7.52	12.70	21.01
DP RCMB (flops/B)	3.76	6.35	7.02

with $E = \Theta(V)$ is $\Theta(V)$. Since the bottom-up approach may need to check all the vertices at each level, the time could be $\Theta(DV)$ (D is the maximum level). Together with the time spent on edge exploration, the computational complexity of the bottom-up approach is $\Theta(DV + E)$. Because we are focusing on real-world graphs, $\Theta(D)$ is extremely small and $E = \Theta(V)$. Thus, the computational complexity of the bottom-up approach is also $\Theta(V)$.

In each level, top-down only travels the vertices in CQ while bottom-up has to travel all graph vertices. Since the average degree of vertices in our graph is constant (e.g. 16), the work for visiting the edges of each vertex (line 8-12 in Algorithm 1 and Algorithm 2) can be considered constant. Therefore, the parallelism of top-down and bottom-up are decided by the loop controls (line 7 in Algorithm 1 and Algorithm 2). If we use a greedy scheduler [13], the work and span of the outer loop control in the bottom-up approach (line 7 in Algorithm 2) is $\Theta(V)$ and $\Theta(lgV)$ respectively. Therefore, the parallelism of bottom-up approach at each level is $\Theta(V/lgV)$. Similarly, the parallelism of top-down approach at each level is $\Theta(V_{CQ}/lgV_{CQ})$ where V_{CQ} is the number of vertices in the Current Queue. Because V is larger than V_{CQ}, bottom-up has higher parallelism than top-down.

B. Bottleneck Analysis

1) Ratio of Computation to Memory Access (RCMA): BFS can be seen as a specific case of Sparse Matrix Vector multiplication (SpMV) [14]. Take $\mathbf{y} = \mathbf{Ax}$ for example, $\mathbf{y}$ is a dense vector that represents NQ, $\mathbf{A}$ is the adjacency matrix of the graph, and $\mathbf{x}$ is a dense vector that represents CQ. $\mathbf{x}(u) = 1$ means vertex u is in the CQ and $\mathbf{x}(u) = 0$ indicates the opposite. $\mathbf{y}(u) \geq 1$ means that vertex u is in the next queue and $\mathbf{y}(u) = 0$ suggests the opposite. As for the sparse matrix $\mathbf{A}$, $\mathbf{A}[u][v] = 1$ means that there is an edge from vertex u to vertex v.

For an $n \times n$ matrix, to complete a matrix-vector multiplication, the processors need to fetch $(n \times n + n)$ elements from the memory. To compute an element of the result vector

Table III
THE BEST SWITCHING POINTS (M) OF DIFFERENT GRAPHS ON CPUs.

$SCALE$	21	21	21	22	22	22	23	23	23
$edgefactor$	8	16	32	8	16	32	8	16	32
Best M	60	114	73	275	258	54	258	97	56

(e.g. $\mathbf{y}(u)$), the processors must do n multiply operations and $n - 1$ add operations. Therefore, the processor has to do $n \times (2n - 1)$ operations to compute $\mathbf{y}$. If an integer is 4 bytes, the RCMA is $\frac{n \times (2n-1)}{4 \times (n \times n + n)} = 0.5$ (computed by Equation (1)).

$$RCMA = \frac{num\_of\_flops\_for\_computation}{num\_of\_bytes\_for\_memory\_access} \quad (1)$$

2) Ratio of Computation to Memory Bandwidth (RCMB): Similar to RCMA (Equation (1)), the RCMB of a specific architecture is defined in Equation (2). Compared to the RCMBs of the evaluated architectures (Table II), the algorithmic RCMA is much lower. For example, the RCMB of Intel Knights Corner MIC is 12.7 while the RCMA of our algorithm is about 0.5, which means the limited memory bandwidth may not match the high processing power required for BFS exploration.

$$RCMB = \frac{theoretical\_peak\_performance}{theoretical\_memory\_bandwidth} \quad (2)$$

C. Influencing Factors of the Best Switching Point

Previous research shows that different graphs have different best switching points on the same platform [4]. After extending the search range of the best switching point (from [1, 30] to [1, 300]), we find that the best switching point changes significantly among different graphs (Table III). We also find that the platforms have a significant impact on selecting the best switching point. For the same graph, using the best switching point of CPUs for GPUs can lead to $2 \times -3 \times$ performance decrease. This is because bottom-up and top-down have different parallelism (section III-A) and memory-access patterns. In general, the best switching point of the combination method is closely related to the graph information and the experimental platform information. Specifically, in our experiment, the graph information includes the number of vertices, the number of edges, and the four statistical parameters used in graph construction (A, B, C, D in Table I). The architecture information includes the peak performance, the memory bandwidth, and the L1 cache size (Figure 7). Because the graph and platform information consist of more than ten parameters in our experiments, it is almost impossible to predict the best switching point manually (e.g. develop a formula). Thus, we use regression to predict the best switching point in real time.

D. Getting the Switching Point through Regression Analysis

To implement the regression method (Section II-C), we need to know what information to include in the training
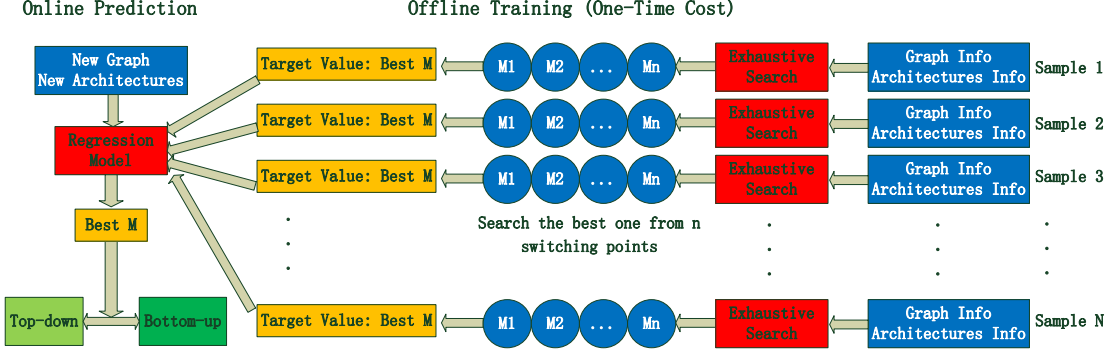
Figure 6. Each sample corresponds to the information of one BFS traversal (graph and architectures). For the off-line approach, we use exhaustive search to get the target value (best M) for each sample. The regression model is then generated through the training based on these samples and their corresponding target values. For the on-line case (at runtime), we use the regression model to predict the best M for the BFS traversal based on the new graph and the new architecture information.

sample X and target value y. As illustrated in Fig. 7, each training sample X_i corresponds to the information of one graph traversal. Specifically, each sample contains the graph information (G_i), top-down architecture information ($Arch - TD_i$), and bottom-up architecture information ($Arch - BU_i$). $Arch - TD_i$ and $Arch - BU_i$ are the same if top-down and bottom-up are on the same architecture. The target value y_i of X_i is the best switching point for exploring G_i on $Arch - TD_i$ and $Arch - BU_i$. For example, suppose the peak performance, L1 cache size, and the memory bandwidth of $Arch - TD_i$ are 512 Gflops, 512 KB, and 100 GB/s respectively. For $Arch - BU_i$, they are 1024 Gflops, 768 KB, and 128 GB/s. The number of vertices, number of edges, A, B, C, and D of G_i is 32 million, 256 million, 0.57, 0.19, 0.19, and 0.05, respectively. The best switching point is 96. In this case, the training sample is (96: 32, 256, 0.57, 0.19, 0.19, 0.05, 512, 512, 100, 1024, 768, 128).

Illustrated in Figure 6, the regression process can be divided into two stages: off-line training and on-line prediction. We can get the model from the training stage and use the model to make predictions in the prediction phase. Although generating a model can be time-consuming, it is a one-time cost. Once we have a model, it can be used for different BFS traversals at runtime.

The training stage can be described by the following steps:

step 1) For a test graph G_i that is explored by top-down on architecture $A - TD_i$ and bottom-up on architecture $A - BU_i$, we run the algorithm repeatedly using all possible switching points (M1, M2 ... Mn in Fig. 6). At the same time we use an exhaustive search to get the best switching point (M) resulting in maximum performance.

step 2) We use G_i, $A - TD_i$, and $A - BU_i$ to build a training sample X_i (Fig. 7). M is the target variable of X_i, which is referred to as y_i.

step 3) We can produce N training samples ($N = 140$ in our experiment) and their target variables. The regression
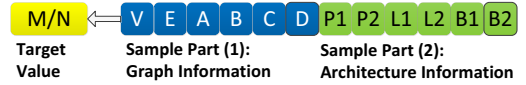

Figure 7. Each training sample contains the graph and the architecture information. V and E are the number of vertices and edges respectively. A, B, C, D are the parameters used in graph construction (Table I). $P1$, $L1$, and $B1$ are the peak performance, L1 cache size, and memory bandwidth, respectively, of the platform that top-down method runs on. $P2$, $L2$, $B2$ are those of the platform that bottom-up method runs on.

model is generated through the training based on these samples and target variables.

More information about this machine learning process can be found in [10]. For on-line prediction at runtime (left part of Fig. 6), the program can use the regression model to predict the best M based on the new sample information. The new sample corresponds to the information of a new graph traversal. The format of this new sample is identical to the format of the training sample (Fig. 7), which includes the information of the new graph, the new top-down and the new bottom-up architectures. The program then uses M as the best switching point for the new graph traversal (Algorithm 3).

E. Effects of the Regression Method

In previous naive combinations ([15], [4]), for a new graph the switching point has to be set manually. From a statistical perspective, regression prediction is more reliable than guessing. Naive combination needs repeated trial-and-error experiments [4]. Although trial-and-error could find a good switching point (90% of the best performance in [4]), it can not be used in practice because the best switching point needs to be searched manually from thousands of possible cases. Moreover, cross-architecture combination requires more complicated switching points, which is extremely hard to do via manual trial-and-error. Automatic trial-and-error
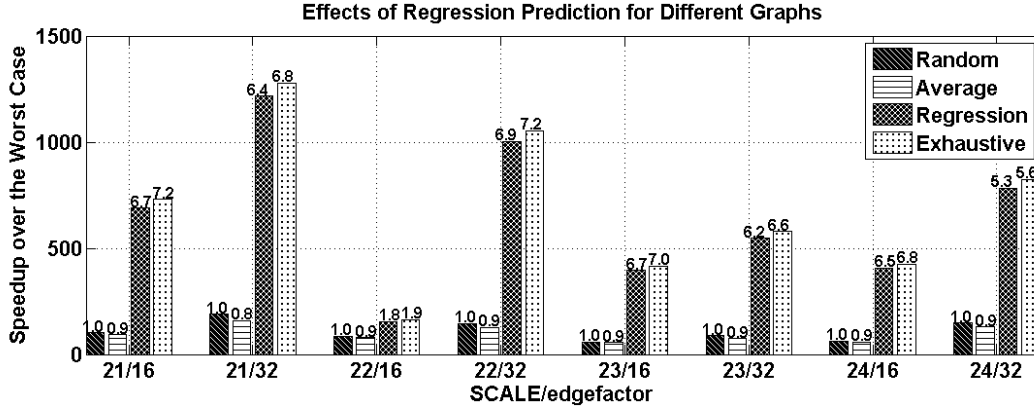
Figure 8. For each graph, the switching points are selected from 1,000 possible cases. *Random* shows the performance when picking the switching point randomly. *Average* represents the average performance over 1,000 switching points. *Regression* shows the performance when using the regression method to predict the switching point. *Exhaustive* shows the performance when the switching point is obtained via exhaustive search, which is the theoretical best. The speedups on the vertical coordinate are over the worst case. The value on top of each bar is the speedup over the *Random* case.

is exhaustive search (hybrid-oracle in [4]), which can get the best solution through searching all the possible cases. However, it can not be used at runtime because it is extremely time-consuming. For example, searching among $1,000$ possible points will at least take $1,000\times$ of BFS execution-time. Compared to exhaustive search, regression prediction is much faster. The execution-time of regression prediction is less than 0.1% of BFS execution-time.

To further evaluate our regression method, we select the switching points from 1,000 possible cases for each graph traversal and summarize the results in Fig. 8. For each graph traversal, three methods are used to select the best switching point: 1) random (*Random*); 2) regression prediction (*Regression*), which is based on 140 training samples; 3) exhaustive search (*Exhaustive*). We also calculate the average performance of these 1,000 switching points (*Average*). In our experiment, the average performance of *Regression* is 95% of *Exhaustive*, which is the theoretical best performance (Fig. 8). The prediction accuracy will be higher with more training samples [13]. The average speedup of *Regression* over *Random* is $6\times$. On the other hand, *Regression* has $695\times$ and $7\times$ speedup over the worst switching point and *Average* (Fig. 8), which means a mistuned switching point can have a significant influence on the overall performance for cross-architecture combination. Therefore, the regression technique can get perfect performance with little runtime overhead.

IV. CROSS-ARCHITECTURE COMBINATION

We first do combination on a single architecture (CPUs, GPUs, and MIC). The combination technique for graph traversal performs better on GPUs compared to CPUs. Take a graph with 8 million vertices and 128 million edges as an example (Table IV), the combination technique (GPUCB) achieves speedups of $16.5\times$ and $15.7\times$ over top-down (G-

PUTD) and bottom-up (GPUBU), respectively. On the CPU, the speedup is $3.4\times$ and $2.8\times$ over top-down (CPUTD) and bottom-up (CPUBU) respectively. There are two main reasons behind this.

From Table IV, we find that 97% of GPUBU time is spent on the first two levels, which is the main reason behind the lower performance compared to CPUBU. In the first level, only the source vertex is in CQ (line 1 in Algorithm 2). For a better bottom-up implementation, we use the CSR (Compressed Sparse Row) format [4] to store the graph and use bitmap [16] for the CQ. In this case, each vertex has to visit almost all of its edges to decide whether the source vertex is its neighbor or not. Therefore, bottom-up has to fetch all the data (vertices and edges) from memory in the first level (line 7-9 in Algorithm 2). As mentioned in section III-B2, the RCMA of BFS is much lower on the RCMB of our architectures. Because of being memory-bound, higher architectural RCMB will intensify the mismatch between the application and architecture. Thus, GPUBU pays a severe penalty.

Table IV shows that 99.7% of GPUTD time is spent on the middle four levels (level 2-5). This is similar to CPUTD, which spends 98.5% of the time on the middle four two levels (level 2-5); the reason is elaborated in Section II-B). For both GPUTD and GPUBU, the time spent on each level is extremely imbalanced. However, this characteristic also makes GPU a good candidate for the combination technique.

In the following we further analyze the combination on GPU (GPUCB) and the combination on CPU (CPUCB) (Table IV). In the first two steps, both GPU and CPU use top-down, where the CPU has $11\times$ speedup over GPU. From level 3 to 7, both GPU and CPU use bottom-up and the GPU achieves $3\times$ speedup over CPU. As mentioned in section III-A, this is because bottom-up provides higher parallelism and thus is more suitable for the massive lightweight threads

Table IV
STEP-BY-STEP OPTIMIZATION, LEVEL TIME IS MEASURED IN SECONDS, THE EVALUATED GRAPH HAS 8 MILLION VERTICES AND 128 MILLION EDGES.
TD: TOP-DOWN, BU: BOTTOM-UP, CB: COMBINATION OF TOP-DOWN AND BOTTOM-UP.

Approach	GPUTD	GPUBU	GPUCB	CPUTD	CPUBU	CPUCB	CPUTD+GPUBU	CPUTD+GPUCB
Level 1 Time	0.000230	**0.438904**	0.000230 TD	0.000779	**0.053730**	0.000728 TD	0.002151 CPUTD	0.002239 CPUTD
Level 2 Time	**0.157750**	**0.131876**	**0.021164 TD**	0.001945	**0.032186**	0.001208 TD	0.002731 CPUTD	0.002608 CPUTD
Level 3 Time	**0.155881**	0.010673	0.008493 BU	**0.074355**	0.015300	0.015643 BU	0.005293 GPUBU	0.005922 GPUBU
Level 4 Time	**0.261753**	0.002783	0.002675 BU	**0.072465**	0.012448	0.011732 BU	0.002288 GPUBU	0.002424 GPUBU
Level 5 Time	**0.044015**	0.001590	0.001600 BU	**0.011941**	0.006933	0.006914 BU	0.001653 GPUBU	0.001658 GPUBU
Level 6 Time	0.000882	0.001474	0.001502 BU	0.000980	0.005121	0.005515 BU	0.001601 GPUBU	0.001596 GPUBU
Level 7 Time	0.000233	0.001468	0.001498 BU	0.000705	0.004987	0.005406 BU	0.001602 GPUBU	0.000286 GPUTD
Level 8 Time	0.000229	0.001466	0.000237 TD	0	0.004972	0.000716 TD	0.001599 GPUBU	0.000234 GPUTD
Level 9 Time	0	0.001466	0.000230 TD	0	0	0	0	0.000230 GPUTD
Total Time	0.620973	0.591701	0.037629	0.163170	0.135677	0.047862	0.018918	0.017196
Speedup	1.0×	1.1×	16.5×	3.8×	4.6×	13.0×	32.8×	36.1×

Table V
SPEEDUPS OF CPUTD+GPUCB OVER GPUTD FOR CERTAIN GRAPHES

$|V|$	2M	2M	2M	4M	4M	4M	8M
$|E|$	32M	64M	128M	64M	128M	256M	128M
Speedup	44×	75×	155×	37×	35×	67×	36×

on GPUs. Since 57% of GPUCB time is spent on the first two levels, using CPUTD to replace GPUTD in the first few levels is extremely necessary for performance improvement. Thus, the two BFS approaches are combined across the architectures, which allows CPU to do top-down and GPU to do bottom-up (CPUTD+GPUBU in Table IV). CPUTD+GPUBU achieves 32.8× speedup over GPUTD.

At levels 8 and 9 of GPUCB and CPUCB, both GPU and CPU switch back to top-down. However, GPU becomes faster than CPU. We believe this is bacause of the low number of vertices and edges in the CQ since processors do not have to fetch a large amount of data from memory. In the compute-intensive scenario, with stronger processing power and memory bandwidth, GPU has certain advantages over CPU. Therefore, it is meaningless for the CPU+GPU solution to switch back to CPU in the last levels. For better performance, the CPU+GPU solution switches from GPUBU to GPUTD in the last few levels since GPUTD is faster than GPUBU when the number of vertices and edges is small (Table IV). Our best solution is CPUTD+GPUCB, which achieves from 35× to 155× (average is 64×) speedup over GPUTD for a series of test graphs (Table V). The CPU-GPU cross-architecture combination achieves 8.5×, 2.6×, and 2.2× average speedup over the MIC, CPU, and GPU combination respectively (Figure 9). This proves that the cross-architecture combination is necessary for performance improvement. The CPUTD+GPUCB solution is described in Algorithm 3.

Algorithm 3: CPU + GPU Combination

Input: Graph Information (GI)
      CPU Information (CPUI)
      GPU Information (GPUI)

1   $(M_1, N_1) \leftarrow$ RegressionModel(GI, CPUI, GPUI)
2   $(M_2, N_2) \leftarrow$ RegressionModel(GI, GPUI, GPUI)
3   BFS Initialization
4   **while** *ture* **do**
5   **if** $CQ = \emptyset$ **then**
6   **break**
7   **else**
8   calculate $|E|_{cq}$ and $|V|_{cq}$
9   **if** $|E|_{cq} < |E|/M_1$ *and* $|V|_{cq} < |V|/N_1$ **then**
10   do the top-down on CPU
11   **else**
12   **while** *ture* **do**
13   **if** $|E|_{cq} < |E|/M_2$ *and* $|V|_{cq} < |V|/N_2$ **then**
14   do the top-down on GPU
15   **else**
16   do the bottom-up on GPU
17   **if** $CQ = \emptyset$ **then**
18   **break**
19   **else**
20   calculate $|E|_{cq}$ and $|V|_{cq}$

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. Implementation Details

We use the CSR (Compressed Sparse Row) format to store the graph and bit-map or bool-map to store the queue vector. The compilers are CUDA 5.5 and icc 14.0.2. Multi-threading
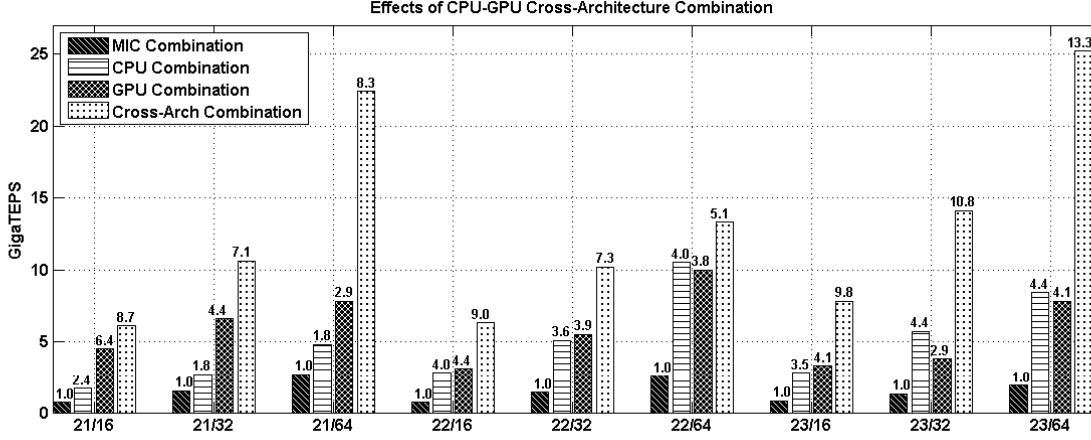
Figure 9. This figure shows the performances for different graphs achieved by different versions of combinations. For each graph, the number of vertices is 2^{SCALE} and the number of edges is $edgefactor \times 2^{SCALE}$. The value on top of each bar is the speedup over the MIC combination.

on CPUs and MIC are based OpenMP. The implementations are evaluated based on the R-MAT graph used in the Graph 500 benchmark [11]. The R-MAT graph is a scale-free graph generated by the Kronecker generator. The graph is divided into four partitions. The initial graph is empty, and edges are added to the graph one by one. Each edge selects one of the four partitions with probabilities A, B, C and D. To generate a specific kind of graph, the users need to set the parameters A, B, C, and D. In our experiment, we set $A = 0.57$, $B = 0.19$, $C = 0.19$, and $D = 0.05$ respectively. The random numbers used in Fig. 8 are based on the rand() function of C stdlib.h library.

B. Strong and Weak Scaling

The strong scaling results (Figure 10(a)) show that performance grows with increasing number of cores. Since the larger graphs in the weak scaling test generally increase the usage of computation units (from one core to multiple cores), it is beneficial to reduce the memory-bound overhead. Thus, our implementation obtains a good weak scaling (Figure 10(b)).

C. MIC Performance

In our experiments, the 8-core single socket CPU has an average $3.3\times$ speedup over the 60-core MIC. Since both CPU and MIC show good strong scaling (Fig. 10(a)), we believe the reason behind the performance gap between CPU and MIC is the difference between their serial versions, which is decided by the single core capacities. A MIC core is much simpler compared to a Sandy Bridge core because it is based on the Intel P54 (the first generation of Pentium) micro-architecture. Take the graph with 4 million vertices as an example, the serial version on CPU has a $20.6\times$ average speedup over MIC for a variety of edgefactors (16, 32, 64). We think that the significant difference comes from
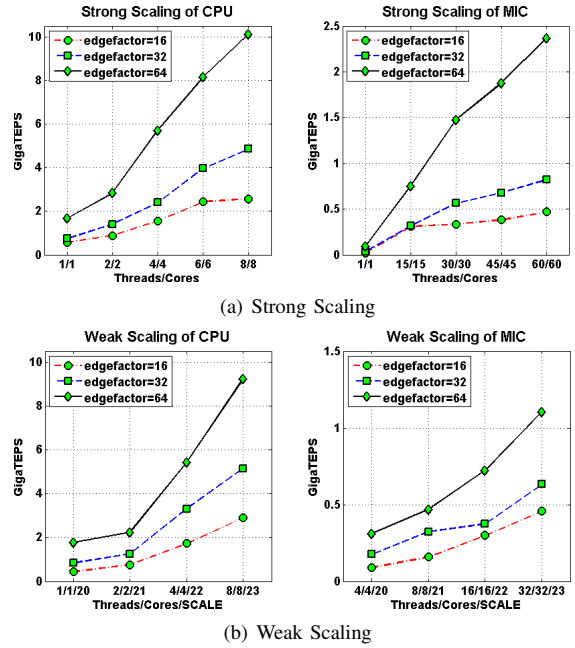

(a) Strong Scaling

(b) Weak Scaling

Figure 10. (a): The results in this figure are based on the graph with 4 million vertices (SCALE=22) and varied edges ($4 \times edgefactor$ million. (b): Each CPU core loads 1 million vertices and $edgefactor$ million edges. Each MIC core loads 0.25 million vertices and $0.25 \times edgefactor$ edges. As the number of cores increases, the total workload increases while the workload of each core remains the same.

three major factors: the first is the $2\times$ clock rate difference between the CPU and the MIC; the second is that the MIC core cannot execute two instructions from the same thread in consecutive cycles, which would add another factor of 2; the third reason is the absence of an L3 cache and the lack of support for out-of-order execution in the MIC core, which accounts for another factor of 5. One the other hand, we use

Table VI
AVERAGE PERFORMANCES FOR DIFFERENT DATA SIZE ON DIFFERENT
ARCHITECTURES (GTEPS).

Architectures	2M vertices	4M vertices	8M vertices
CPU/GPU/MIC	3.06/6.32/1.64	6.14/6.23/1.55	5.66/5.00/1.33

the same source code for CPU and MIC without specific optimizations for MIC. SIMD does lead to performance enhancement in our approach because it greatly increases the number of edges to travel. Thus, we abandon the SIMD optimization for the major computation part. This, however, may have the most impact on MIC because MIC is 512-bit SIMD, which would have been a unique advantage. This maybe another reason why the performance of MIC is much lower than CPU and GPU.

D. Comparison against other implementations

The highest published performance on CPUs and GPUs is achieved by Beamer et al. [4]. Our approach achieves an average $1.12\times$ speedup over theirs on similar architectures (16-core Sandy Bridge CPUs) for R-MAT graphs. However, this is not our major contribution, we only want to justify that our CPU implementation is state-of-art. Beamer's switching points are obtained through trial-and-error and exhaustive search, which can not be used in practice. The highest published performance on MIC is reported by Gao et al. [17]. Their best reported performance is 0.14 GigaTEPS for a graph with 64 million vertices and 1024 million edges. We achieve a $13\times$ speedup for the same graph and on the same platform. The Graph 500 benchmark also provides parallel implementation source codes, we run them on an 8-core CPU platform to provide a point-to-point comparison. Our CPU implementation achieves $4.96 - 21.0\times$ (average is $11.0\times$) speedups over theirs.

These comparisons justify that our regression-analysis approach is effective on different architectures. The additional speedups achieved by adding the cross-architecture technique justify the added optimizations are highly efficient. For example, our cross-architecture combination achieves $16.4 - 63.2\times$ (average is $29.3\times$) speedups over the Graph 500 implementations.

VI. RELATED WORK

We [18] previously proposed a level synchronized parallel algorithm based on Cray MTA-2, which makes full use of the massive fine-grained threads and low overhead synchronization provided by the system. Merrill et al. [19] achieved a fine-grained parallelization through efficient prefix sum on GPUs. Leiserson and Schardl [20] designed an original multi-set data structure, called bag, to replace the conventional FIFO queue. Agarwal et al. [16] developed an efficient multi-socket algorithm with optimizations on the memory locality and cache utilization. Chhugani et al. [21]

did a series of architectural optimizations (e.g. lock-free, atomic-free, vertices rearrangement) to maximize the single-node efficiency on a dual-socket CPUs platform. Li et al. [22] proposed a runtime system able to dynamically transition between different implementations on GPUs. Nasre et al. [23] designed a hybrid approach of data-driven and topology-driven for graph algorithms on GPUs.

Beamer et al. [4] and Hong et al. [15] are the closest work to ours. In [4] the authors apply a combination of top-down and bottom-up approaches only on CPU. In [15] the authors apply the combination of purely top-down methods on CPU and also use the same combination strategy on GPU. In our approach, the hybrid method applies the top-down method on CPU and the bottom-up/top-down mixed method on GPU. To our knowledge, this is the first attempt to use different architectures for combining the top-down and bottom-up. Compared to trail-and-error or exhaustive search based heuristics (e.g. hybrid-oracle method in [4]) in their work, more importantly, we propose a fast and accurate switching strategy based on regression. The regression technique is able to put the combination method into practice since it ensures perfect performance (at least 95% of best performance with 140 training samples) and brings little runtime overhead (less than 0.1% of BFS execution-time).

VII. CONCLUSION

In order to get the best switching point automatically in real time, we propose a combination technique based on regression analysis, which is much more convenient and time-efficient compared to previous trail-and-error or exhaustive search based approaches. Our approach can achieve $695\times$ speedup over the worst switching point and only increases less than 0.1% of the execution time while the exhaustive search may increase the execution time a thousand times. Furthermore, our cross-architecture combination efficiently uses the popular heterogeneous platforms and greatly improves the performance of BFS, and the average additional speedup is $3\times$ behind different architectures.

Although the flops peak performance of GPUs is much better than that of CPU, CPUs achieves better performance for graphs with large data sizes. This is because CPU is equipped with a more matchable memory bandwidth than the computation-intensive GPUs. For MIC, the lower clock rate, constrained instruction-execution scheme, and reduced cache size make the performance of the serial version much worse than that of CPU, which is one of the major reasons for the low overall performance.

VIII. ACKNOWLEDGE

REFERENCES

[1] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 205–218.

[2] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani, "Global protein function prediction from protein-protein interaction networks," *Nature Biotechnology*, vol. 21, no. 6, pp. 697–700, 2003.

[3] Y. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *IEEE/ACM International Conference on Computer-Aided Design, 2009. ICCAD 2009.* IEEE, 2009, pp. 539–546.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 1–10.

[5] N. R. Draper, H. Smith, and E. Pownell, *Applied regression analysis*. Wiley New York, 1966, vol. 3.

[6] J. Dongarra. (June 2013) Top500 list. [Online]. Available: http://www.top500.org/lists/2013/06/

[7] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[8] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th International Conference on Machine Learning*. ACM, 2008, pp. 104–111.

[9] Y. You, S. Song, and H. Fu, "MIC-SVM: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures," in *2014 IEEE 28th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2014, pp. 809–818.

[10] C. Lin. (2014) Libsvm – a library for support vector machines. [Online]. Available: http://www.csie.ntu.edu.tw/ cjlin/libsvm/

[11] Graph500. (2013) Graph500. [Online]. Available: http://www.graph500.org/

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, 3rd edition," *The MIT Press*, 2009.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[14] U. V. Catalyurek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *ACM/IEEE 2001 Conference on Supercomputing*. IEEE, 2001, pp. 42–42.

[15] 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*. IEEE, 2011, pp. 78–88.

[16] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[17] G. Tao, L. Yutong, and S. Guang, "Using MIC to accelerate a typical data-intensive application: the breadth-first search," in *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 2013, pp. 1117–1125.

[18] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *International Conference on Parallel Processing. ICPP 2006.* IEEE, 2006, pp. 523–530.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[20] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the non-determinism of reducers)," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 303–314.

[21] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 378–389.

[22] D. Li and M. Becchi, "Deploying graph algorithms on GPUs: An adaptive solution," in *2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2013, pp. 1013–1024.

[23] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2013, pp. 463–474.