Cache-efficient Graph Cuts on Structured Grids

Ondřej Jamriška Daniel Sýkora
Czech Technical University in Prague, FEE
{jamriond,sykorad}@fel.cvut.cz

Alexander Hornung
Disney Research Zurich
hornung@disneyresearch.com

Abstract

Finding minimal cuts on graphs with a grid-like structure has become a core task for solving many computer vision and graphics related problems. However, computation speed and memory consumption oftentimes limit the effective use in applications requiring high resolution grids or interactive response. In particular, memory bandwidth represents one of the major bottlenecks even in today's most efficient implementations.

We propose a compact data structure with cache-efficient memory layout for the representation of graph instances that are based on regular N-D grids with topologically identical neighborhood systems. For this common class of graphs our data structure allows for 3 to 12 times higher grid resolutions and a 3- to 9-fold speedup compared to existing approaches. Our design is agnostic to the underlying algorithm, and hence orthogonal to other optimizations such as parallel and hierarchical processing. We evaluate the performance gain on a variety of typical problems including 2D/3D segmentation, colorization, and stereo. All experiments show an unconditional improvement in terms of speed and memory consumption, with graceful performance degradation for graphs with increasing topological irregularities.

1. Introduction

Minimal cuts on graphs have been studied over decades [12] and have developed into a fundamental solution to problems in various disciplines. In computer vision and graphics, applications range from segmentation [4, 24, 28], stereo and shape reconstruction [7, 17, 35], editing and synthesis [1, 23, 29, 32], fitting and registration [5, 13, 22] to pose estimation and more [20].

Typically, in those application domains the underlying graph structures can be characterized as regular N-D grids, where all nodes have topologically identical neighborhood systems, *i.e.*, each node is connected in a uniform fashion to all other nodes lying within a given radius (Fig. 1). One of the algorithms which are efficient on such graph constructions is the popular method by Boykov and Kolmogorov (BK) [6].

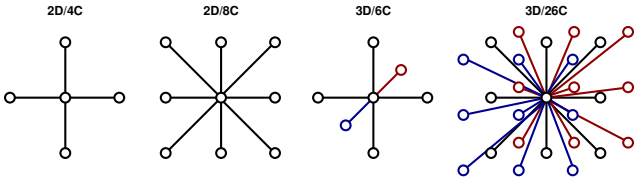

Figure 1: Typical examples of neighborhood connectivity.

Central to this method is the concept of the residual graph; based on the seminal Ford-Fulkerson's algorithm [12], the BK algorithm performs a bi-directional search for augmenting paths on the residual graph, together with a tree-reuse strategy which avoids rebuilding the search trees from scratch after each augmentation. Although its worst case complexity may be worse than alternative algorithms it was shown [6,8,11] that, for the specific graph structures encountered in vision and graphics, BK typically outperforms other approaches such as the push-relabel algorithm [15] and its more recent variants [14].

However, despite many advances, using min-cuts for processing high resolution data at sufficient speed is still a significant challenge. For example, in the previously mentioned application domains, segmenting and editing HD video in real-time or high resolution 3D model reconstruction from megapixel images remain very active fields of research. Recent research on parallelization [25] and strictly polynomial extensions [16] shows that there is still considerable need and room for improvement of the BK algorithm.

In this paper we show that significant further improvements can be achieved by optimizations of the memory bandwidth. Our main contribution is a compact data structure specifically designed for an efficient encoding of commonly used structured N-D graphs, and, in particular, their residual graphs. We exploit the typical connectivity patterns observed in the previously mentioned application domains for a highly efficient memory layout. Our method allows for 3 to 12 times higher grid resolutions and achieves a 3- to 9-fold performance gain. Our optimizations are agnostic

of the actual implementation and hence complementary to other types of min-cut algorithms. Moreover, our method does not strictly rely on the graph regularity assumptions, and shows superior performance on graphs with moderate amount of topological irregularities as well.

2. Previous work

The body of work on optimization strategies for graph cut computation comprises a variety of fundamentally different approaches, including hierarchical approximations [18, 26, 27], capacity scaling [19], strategies for reusing computations [20], or parallel processing [10, 25, 30, 31, 34]. We therefore concentrate our review on those works related to cache-efficient data structures that are closest related to our paper.

One of the first approaches specifically focusing on memory layout and cache optimizations is the method by Bader and Sachdeva [3]. Their focus is on a parallel implementation of the push-relabel method [15] for general, irregular graphs on multi-processor machines with shared memory. For each edge they store its capacity, current flow and also link to its reverse edge to reduce the number of memory accesses when the reverse edge is just read and not updated. They also use contiguous allocation of memory for parallel pairs of edges to ensure spatial locality of the data structures during the updating. Delong and Boykov [10] employed similar strategies and presented a different parallel variant of the push-relabel approach that targets the specific grid-like graph structures encountered in computer vision. A particular strength of their approach is the control over locality of memory accesses that allows for more efficient processing of very large graph structures. Liu and Sun [25] describe a parallel implementation of the BK algorithm, using a grid-like partitioning to maintain locality of computations within subgraphs. Their approach is implicitly cache-friendly and partially achieved super-linear speedups on shape fitting problems. Recently Goldberg et al. [16] proposed several low-level optimizations to avoid cache violating access. They also replace adjacency lists with arrays and reported that this optimization resulted on average in a 20% performance gain over the original BK implementation.

Complementary to these works we show that the optimization of *memory bandwidth* is a further important resource for optimization, gaining significant improvements in terms of speed *and* memory utilization.

3. Our approach

Computation of maximum flow on graphs typically requires frequent data transfers between memory and CPU causing significant latencies in the run time of max-flow algorithms. Our aim is to ease this memory bandwidth bottleneck by employing a compact graph representation with cache-friendly memory layout that exploits the regular structure of grid-like graphs.

3.1. Compact representation of the residual graph

Central to most max-flow algorithms [6, 10, 16] is the so called *residual graph* which maintains the distribution of flow during the algorithm run time. This data structure requires constant access and updates, as such its compact representation is crucial in order to achieve a significant reduction in memory bandwidth.

For the encoding of general graphs one usually requires a representation that stores the connectivity information explicitly. These are realized by pointer-heavy data structures (like the adjacency list) which reference elements from separate collections of nodes and edges. The pointers often comprise the majority of the graph's memory footprint, in particular on 64-bit CPUs where single pointer occupies eight bytes.

However, by exploiting prior knowledge of the graph structure we can eliminate the need for pointers altogether by determining connectivity information on the fly. To that end we employ a single 32-bit integer index for addressing individual nodes. Since nodes are arranged in a N-D grid with repeated neighborhood connectivity we can enumerate them in such a way that their neighbors are always at constant offsets. Thanks to this property accessing to neighboring nodes requires only picking the right offset and adding it to the current node's index.

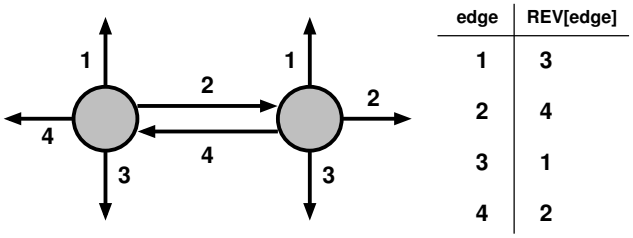

Figure 2: Enumeration of outgoing edges and reverse edge lookup using the REV table.

As the connectivity information is implicit in our representation, there is no need to maintain a separate collection of edges. Instead, we store the information associated with edges in their tail nodes, *i.e.*, each node is equipped with the data of its outgoing edges. Each edge in the residual graph is associated with residual capacity rc, which is the amount of flow that can be further pushed along the edge without exceeding its capacity. In our representation we impose fixed ordering on the node's k outgoing edges and enumerate them with index $1 \dots k$ (see Fig. 2). Each node is then associated with k-tuple $(rc_1, \dots, rc_k)$ of its outgoing edges' residual capacities. This scheme assumes that each

node has the same degree, however, it can be used even in cases when the graph is not perfectly k-connected by assigning zero residual capacity to missing edges or to edges adjacent to missing nodes.

When traversing the residual graph, max-flow algorithms often have to access residual capacities of reverse edges. We facilitate this access by referencing the opposite direction edge in the rc tuple of neighbor's outgoing edges. To determine the index of the opposite direction edge, we use a small lookup table REV (see Fig. 2). When a node lies at the grid boundary, the requested neighbor might not exist. We avoid handling this situation as a special case by extending the grid with a border layer of sentinel nodes whose outgoing edges have residual capacities all set to zero. In this way there always exists a valid neighbor, and referencing of reverse edges can never cause an out-of-bounds access.

In addition to the regular edges between neighboring nodes, each node can be potentially connected to both terminals by s/t links. Similarly to the implementation[1] of [6], we perform a trivial augmentation of the source-node-sink path during the graph initialization and only store the residual capacity $rc_{s/t}$ of the link that remained unsaturated (or set it to zero when both became saturated). In the context of push-relabel algorithms, this value corresponds to the node's initial excess/deficit.

In summary, to represent the residual graph, we maintain a collection of nodes, and for each node we only store the residual capacities of its outgoing edges and the residual capacity of the remaining unsaturated s/t link. In this collection, the nodes are addressed by a 32-bit index and indices of neighbor nodes are computed on the fly. Each node has the same number of outgoing edges, and the edges have a fixed ordering which allows quick determination of the opposite direction edge using a lookup table. This approach is similar to representations employed by [25], the implementation[2] of [10], and the GPU oriented graph encoding of [34], but crucially, we further combine it with structure splitting and blocked array reordering to achieve even better utilization of caches, as described in the following sections.

3.2. Structure splitting

Besides residual capacities max-flow algorithms usually introduce additional algorithm-specific fields. For example, BK maintains two search trees with marking heuristic [21] requiring for each node to store the tree membership tag, a reference to a parent node, an estimation of the distance to its terminal, and a time-stamp indicating when the distance was computed. However, from the optimization standpoint the key observation is that these fields are typically not accessed at the same time during the max-flow computation. For instance, algorithms based on augmenting paths alternate between search and augmentation while accessing different subsets of fields in each stage. In literature on cache optimization [9] these frequently accessed fields are typically referred to as *hot* while unused fields are *cold*. To improve cache utilization we want to preserve space for *hot* fields and avoid transfers of *cold* fields. This can be achieved by rearranging the node information using the structure-of-arrays layout (SoA) [9]. Instead of storing all nodes in a single array of structures with all fields packed together, we split the individual fields into separate arrays. With this layout the data can be naturally split into *hot* and *cold* portion.

3.3. Cache-friendly layout of arrays

We observe that even though the data access pattern of max-flow algorithms is irregular, it usually exhibits a certain amount of spatial locality. For example, when some node is visited during the computation, it is likely that other nearby nodes will be visited afterwards. The aim is to exploit this behavior to further improve utilization of caches. As transfers between cache and memory are executed in blocks (*cache lines*) with fixed size (typically 64 bytes) we can rearrange arrays into a *blocked layout*. Here all blocks have a size equal to the size of cache lines and their fields are arranged in such a way that spatially close nodes on the grid become spatially close in memory (see Fig. 3). When a field from the block is accessed for the first time, a cache miss occurs and the field is loaded into the cache along with fields of other nodes lying in the same block. Thanks to this behavior fields of nearby nodes can be quickly accessed in further steps of the algorithm.

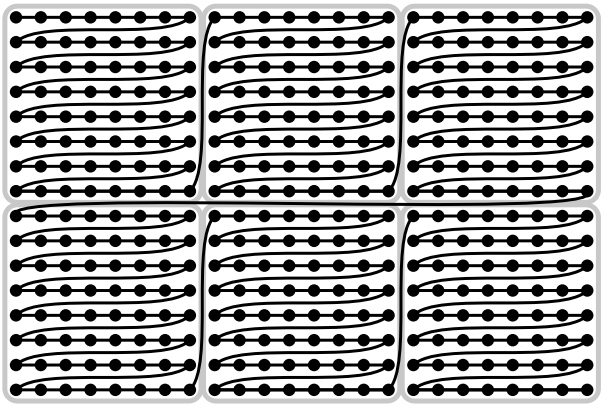

Figure 3: Blocked array layout.

The only issue associated with this rearrangement is an efficient computation of a node's neighbor indices. A naive approach would be to reconstruct the grid coordinates back from the nodes's index, translate them by the desired offset and then recompute a neighbor's index from the new coordinates. However, this computation can be very costly

[1] vision.csd.uwo.ca/code/maxflow-v3.01.zip
[2] vision.csd.uwo.ca/code/regionpushrelabel-v1.03.zip

requiring several division and modulo operations. To keep the index computation cheap, we consider only blocks with power of two extents (not necessarily equal) and with scanline ordering of nodes inside the block as well as the blocks themselves (see Fig. 3). Grids with dimensions that are not evenly divisible by block size are padded by dummy nodes. In this setting coordinates inside the block can be easily separated from the node's index by bit-mask operations. This helps to quickly distinguish between different neighbor offsets for nodes inside a block. We demonstrate this computation for 2D 4-connected grids in Appendix, other common neighborhood systems (2D8C, 3D6C, and 3D26C) are listed in the supplementary PDF.

3.4. Implementation details

To demonstrate the effectiveness of our method, we incorporated the proposed optimizations in our own re-implementation of BK which we call OBK. The source code of OBK is available at *http://gridcut.com*. Our implementation always uses the full combination of the compact residual graph representation (CR), the structure of arrays (SoA), and the blocked array layout (BLK).

Because the growth and adoption stages of BK access residual capacities only to determine whether the edges are saturated or not, we found it beneficial to store the saturation status of a node's k outgoing edges in an additional k-bit field. Each bit in this field indicates whether the corresponding edge has non-zero residual capacity. Even though the bit field must be updated whenever the edge saturates or desaturates (in the augmentation stage), the overall benefit of fetching less data from memory more than compensates for the overhead of updates. The use of a saturation bit field is advantageous mainly on denser graphs with large capacities. We used it in all our experiments except for the 4-connected grids which are not dense enough to amortize its maintenance.

Our implementation also employs the "marking" heuristic described in [21], except for 4-connected grids where we observed an average 7% increase in algorithm run time when the heuristic was enabled. However, we still make use of the timestamping mechanism to mark the nodes whose origin was already traced to one of the terminals. In its original form the heuristic is applied in both the growth and the adoption stages of BK. We use a slightly different approach where we apply the heuristic only during adoption stage, as in our experiments the path length reduction achieved by parent reassignment in the growth stage did not account for its overhead, and on average increased the run time by 9%.

4. Results and Discussion

To verify the efficiency of our method we performed an exhaustive benchmark (see Tab. 1) which combines graphs from well-known data sets provided by The University of Western Ontario[3] and Middlebury College[4] (α-expansion [7, 33]). These graphs originate from various applications such as restoration (1-2), stereo (3-8), 2D segmentation (9-11), photomontage (12-13), 3D shape fitting (21-22), and 3D segmentation (23-48). In addition to that we also include graphs (14-20) used to solve the colorization problem [32] which is similar to 2D image segmentation [4], but with only very sparse links to the s/t terminals. Besides various applications and density of s/t links we also provide variability in grid dimension (2D/3D), number of nodes (20k-12M), neighborhood topology (4/6/26-connected), and number of bits needed to store the maximal capacity (8/16/32-bit).

To perform the comparison we downloaded the latest implementation[1] of BK and used our optimized implementation OBK. We compiled both codes in 32-bit and 64-bit modes with identical compiler settings and ran them on various CPUs (Xeon & Core i3/i7) with different cache sizes (3/6/8M). Exact configurations as well as used compilers and switches are listed in Tab. 1. For each graph instance we selected an appropriate code template having an optimal data type able to store the maximal capacity in the graph (same for BK and OBK). In contrast to previous benchmarks we decided to measure processing time not only for maximum flow computation, but also for graph initialization and output phase. This is crucial for a fair comparison with BK as in OBK we have to perform initial data rearrangement which takes some additional CPU time.

The resulting memory footprint reductions as well as speedups on selected CPUs are listed in Tab. 1. As shown in the table, the memory footprint reduction increases with the number of nodes and graph connectivity. It ranges from 2x to 6x for 32-bit and 3x to 12x for 64-bit modes. The average speedup over all instances and CPUs is 2.7x for 32-bit and 4.4x for 64-bit modes. However, as is visible from the table and graphs in Fig. 4, there is a notable variability. Despite of this fact, we found a few interesting trends which are worth mentioning.

First we observed that the main performance gain is always caused by the compact data representation (CR on the bottom graph in Fig. 4). The importance of the lower memory consumption is also apparent when comparing 32-bit with 64-bit modes (top and middle graph in Fig. 4). Today 64-bit mode is preferred in practice as it allows for a significantly larger addressing space. On the other hand it requires 64-bits to store all the pointers used in the BK method, therefore a larger amount of data needs to be stored/transferred compared to OBK where only a few array pointers are needed and all nodes are indexed by 32-bit integers. Another important factor which influences speedup is the size of frequently accessed area in the memory. The

[3] http://vision.csd.uwo.ca/maxflow-data/
[4] http://vision.middlebury.edu/MRF/

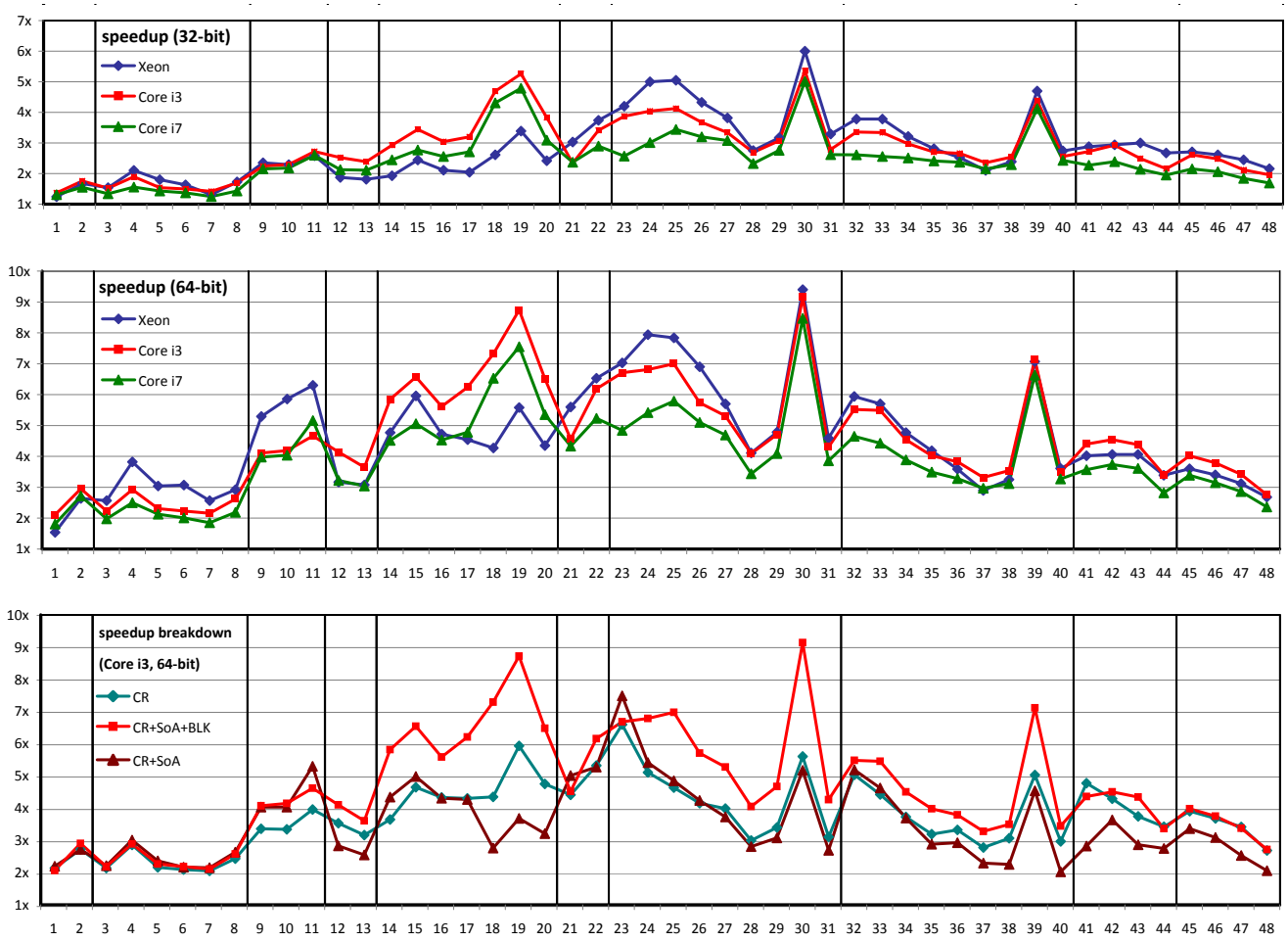

Figure 4: Speedups of OBK with respect to BK in 32-bit (top) & 64-bit (middle) modes and speedup breakdown (bottom).

performance gain is only moderate when such area is so small that it can be fully fitted into the cache (see speedups for small instances 1-8 on all CPUs). Due to same reason cache-efficient memory layout (CR+SoA+BLK, bottom in Fig. 4) brings notable improvements only for larger instances (9-48). It is also interesting to note that the structure splitting technique alone (CR+SoA) typically does not improve the final performance over CR. Only in combination with the blocked layout (BLK) we see notable speedups. Another question is whether cache size influences the speedup. There is not a straightforward dependance, however, from the graphs displaying the difference between Core i3 (3M) and Core i7 (8M) (see Fig. 4) one may deduce that our approach brings better speedups on CPUs with smaller caches.

As OBK brings significant performance gain for a variety of vision and graphics problems we decided to compare it directly with existing state-of-the-art max-flow/min-cut methods. We selected two recent algorithms: (1) voronoi-based pre-flow push (VPP) [2] and (2) incremental breadth-first search (IBFS) [16] which to our best knowledge report best speedups over the original BK. Based on a DLL provided by authors of VPP we performed our own comparison with BK and found that the results shown in [2] were affected by three important factors. Firstly the authors did not set the graph for the BK algorithm correctly and introduced many redundant edges with zero capacities. Secondly, they did not use the latest version of BK, and also did not use full compiler optimizations (`/Ox` in Visual Studio). These points are crucial for the peak performance of BK. With all these issues fixed, VPP performed only comparable or even worse than BK. The performace comparison of VPP with respect to OBK is provided in the supplementary material. Regarding IBFS we used the publicly available implementation[5] and found that on all instances OBK achieves notably better absolute times mainly due to inefficient preprocessing phase used in IBFS. The absolute times were often better even when measuring only the computation of maximum flow (see supplementary material for detailed evaluation).

[5] `http://www.cs.tau.ac.il/~sagihed/ibfs/`

	Intel Xeon E5440 @ 2.83 GHz 6M cache | gcc 4.4.0, -O3 -march=native -mtune=generic -DNDEBUG Intel Core i3 M370 @ 2.40 GHz 3M cache | gcc 4.5.2 (32b) / 4.7.0 (64b), -O3 -march=native -mtune=generic -DNDEBUG Intel Core i7 950 @ 3.07 GHz 8M cache | gcc 4.5.2 (32b) / 4.7.0 (64b), -O3 -march=native -mtune=generic -DNDEBUG				Memory		Xeon		Core i3		Core i7	
					32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
	instance	topo	# nodes	captype	reduction	reduction	speedup	speedup	speedup	speedup	speedup	speedup
1	/mid/denoise/penguin	2D/4C	22k	16-bit	1.71	3.11	1.24	1.54	1.37	2.11	1.31	1.81
2	/mid/denoise/house	2D/4C	66k	32-bit	1.49	2.72	1.67	2.64	1.76	2.95	1.55	2.72
3	/mid/stereo/tsukuba	2D/4C	111k	8-bit	1.91	3.64	1.54	2.57	1.52	2.23	1.34	1.98
4	/uwo/stereo/BVZ-tsukuba	2D/4C	111k	8-bit	1.91	3.64	2.10	3.82	1.90	2.93	1.56	2.50
5	/uwo/stereo/BVZ-sawtooth	2D/4C	165k	8-bit	1.95	3.73	1.80	3.04	1.54	2.31	1.43	2.13
6	/uwo/stereo/BVZ-venus	2D/4C	166k	8-bit	1.93	3.68	1.63	3.07	1.50	2.23	1.37	2.01
7	/mid/stereo/venus	2D/4C	166k	16-bit	1.85	3.37	1.31	2.57	1.42	2.16	1.25	1.85
8	/mid/stereo/teddy	2D/4C	169k	8-bit	1.97	3.76	1.73	2.92	1.70	2.62	1.43	2.19
9	/mid/segment/flower	2D/4C	270k	16-bit	1.87	3.41	2.35	5.29	2.24	4.11	2.15	3.98
10	/mid/segment/person	2D/4C	270k	16-bit	1.87	3.41	2.29	5.86	2.30	4.19	2.18	4.04
11	/mid/segment/sponge	2D/4C	307k	16-bit	1.86	3.40	2.58	6.30	2.72	4.66	2.60	5.16
12	/mid/photomontage/family	2D/4C	426k	32-bit	1.56	2.86	1.87	3.17	2.52	4.14	2.13	3.22
13	/mid/photomontage/panorama	2D/4C	514k	32-bit	1.55	2.83	1.81	3.08	2.39	3.65	2.11	3.04
14	/ctu/lazybrush/footman	2D/4C	593k	16-bit	1.89	3.44	1.93	4.77	2.93	5.85	2.45	4.52
15	/ctu/lazybrush/hmdman	2D/4C	593k	16-bit	1.89	3.44	2.44	5.96	3.44	6.57	2.77	5.06
16	/ctu/lazybrush/mangadinner	2D/4C	593k	16-bit	1.89	3.44	2.11	4.72	3.04	5.62	2.56	4.53
17	/ctu/lazybrush/mangagirl	2D/4C	593k	16-bit	1.89	3.44	2.04	4.54	3.20	6.24	2.71	4.78
18	/ctu/lazybrush/elephant	2D/4C	2370k	16-bit	1.90	3.48	2.61	4.27	4.69	7.32	4.31	6.53
19	/ctu/lazybrush/bird	2D/4C	2372k	16-bit	1.91	3.48	3.39	5.58	5.26	8.74	4.79	7.55
20	/ctu/lazybrush/doctor	2D/4C	2373k	16-bit	1.91	3.48	2.42	4.35	3.83	6.51	3.09	5.35
21	/uwo/shapefit/LB07-bunny-sml	3D/6C	806k	8-bit	2.51	4.86	3.03	5.60	2.34	4.56	2.37	4.33
22	/uwo/shapefit/LB07-bunny-med	3D/6C	6311k	8-bit	2.70	5.21	3.74	6.53	3.41	6.19	2.90	5.23
23	/uwo/seg3d/bone_subxyz_subxy.n6c10	3D/6C	246k	8-bit	2.31	4.32	4.20	7.03	3.87	6.71	2.57	4.84
24	/uwo/seg3d/bone_subxyz_subx.n6c10	3D/6C	492k	8-bit	2.39	4.46	5.00	7.94	4.04	6.81	3.01	5.42
25	/uwo/seg3d/bone_subxyz.n6c10	3D/6C	983k	8-bit	2.46	4.61	5.05	7.84	4.13	7.00	3.44	5.79
26	/uwo/seg3d/bone_subxy.n6c10	3D/6C	1950k	8-bit	2.53	4.73	4.33	6.90	3.67	5.74	3.20	5.10
27	/uwo/seg3d/bone_subx.n6c10	3D/6C	3899k	8-bit	2.57	4.80	3.82	5.70	3.34	5.31	3.08	4.69
28	/uwo/seg3d/liver.n6c10	3D/6C	4162k	8-bit	2.67	4.99	2.75	4.11	2.69	4.09	2.33	3.44
29	/uwo/seg3d/babyface.n6c10	3D/6C	5063k	8-bit	2.66	4.98	3.17	4.78	3.07	4.71	2.76	4.09
30	/uwo/seg3d/bone.n6c10	3D/6C	7799k	8-bit	2.61	4.88	6.00	9.40	5.36	9.16	5.01	8.47
31	/uwo/seg3d/adhead.n6c10	3D/6C	12583k	8-bit	2.67	4.99	3.29	4.59	2.79	4.30	2.62	3.86
32	/uwo/seg3d/bone_subxyz_subxy.n6c100	3D/6C	492k	8-bit	2.39	4.46	3.78	5.94	3.36	5.52	2.61	4.65
33	/uwo/seg3d/bone_subxyz_subx.n6c100	3D/6C	492k	8-bit	2.39	4.46	3.78	5.70	3.34	5.49	2.56	4.43
34	/uwo/seg3d/bone_subxyz.n6c100	3D/6C	983k	8-bit	2.46	4.61	3.21	4.77	2.97	4.54	2.51	3.89
35	/uwo/seg3d/bone_subxy.n6c100	3D/6C	1950k	8-bit	2.53	4.73	2.81	4.18	2.70	4.02	2.41	3.49
36	/uwo/seg3d/bone_subx.n6c100	3D/6C	3899k	8-bit	2.57	4.80	2.52	3.59	2.66	3.83	2.37	3.28
37	/uwo/seg3d/liver.n6c100	3D/6C	4162k	8-bit	2.67	4.99	2.10	2.90	2.35	3.32	2.15	2.97
38	/uwo/seg3d/babyface.n6c100	3D/6C	5063k	8-bit	2.66	4.98	2.39	3.25	2.54	3.54	2.29	3.12
39	/uwo/seg3d/bone.n6c100	3D/6C	7799k	8-bit	2.61	4.88	4.70	7.07	4.39	7.14	4.12	6.63
40	/uwo/seg3d/adhead.n6c100	3D/6C	12583k	8-bit	2.67	4.99	2.74	3.61	2.56	3.49	2.43	3.27
41	/uwo/seg3d/bone_subxyz_subxy.n26c10	3D/26C	246k	8-bit	5.42	10.65	2.88	4.02	2.72	4.40	2.27	3.57
42	/uwo/seg3d/bone_subxyz_subx.n26c10	3D/26C	492k	8-bit	5.59	10.97	2.94	4.06	2.93	4.54	2.39	3.74
43	/uwo/seg3d/bone_subxyz.n26c10	3D/26C	983k	8-bit	5.76	11.30	3.00	4.06	2.49	4.38	2.14	3.61
44	/uwo/seg3d/bone_subxy.n26c10	3D/26C	1950k	8-bit	5.89	11.57	2.67	3.39	2.16	3.41	1.95	2.82
45	/uwo/seg3d/bone_subxyz_subxy.n26c100	3D/26C	246k	8-bit	5.42	10.65	2.71	3.60	2.61	4.02	2.15	3.39
46	/uwo/seg3d/bone_subxyz_subx.n26c100	3D/26C	492k	8-bit	5.59	10.97	2.61	3.40	2.48	3.79	2.06	3.15
47	/uwo/seg3d/bone_subxyz.n26c100	3D/26C	983k	8-bit	5.76	11.30	2.45	3.12	2.12	3.42	1.84	2.86
48	/uwo/seg3d/bone_subxy.n26c100	3D/26C	1950k	8-bit	5.89	11.57	2.16	2.68	1.96	2.76	1.69	2.36
	average:				**2.77**	**5.26**	**2.81**	**4.58**	**2.84**	**4.65**	**2.47**	**3.99**

Table 1: Detailed performance evaluation for different problem instances.

This leads us to the conclusion that for most vision and graphics problems OBK outperforms current state-of-the-art and delivers best single core performance. However, despite this fact we believe that by incorporating our optimizations into IBFS and making the preprocessing phase more efficient one can possibly beat OBK on selected instances. The key limitation of our approach is that it is tailored to structured grid-like graphs with nodes having identical neighborhood connectivity. However, since we can always introduce redundant edges with zero capacities in places where the edge is missing in the original graph we can handle even more general topologies. To measure the efficiency of our approach in these irregular cases we conducted an experiment where we randomly removed a percentage of edges from the original graphs (22 & 30) having full 6-connected neighborhoods. We then build a new irregular graph on which we run BK as well as OBK with redundant edges, and measured the performance gains. Fig. 5 shows a graceful performance degradation with increasing topological irregularities. However, further investigation is needed to verify that behavior in more practical scenarios.

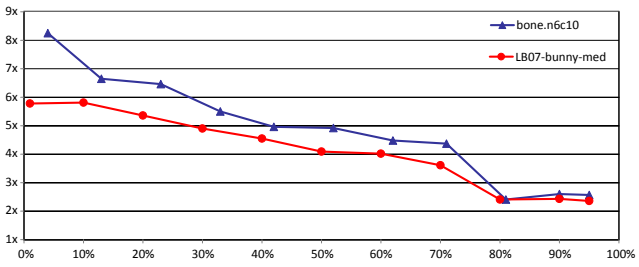

Figure 5: Speedup of OBK with respect to the percentage of removed edges.

5. Conclusions

We have presented a set of cache-efficient optimizations that enable a notable reduction of memory bandwidth when computing graph cuts on structured N-D grids. Our experimental evaluation shows that the proposed optimizations achieve a significant performance gain as well as a reduction of the memory footprint, which renders our method particularly useful for interactive applications and for large problems. Finally, the presented techniques are complementary to other optimizations and can be easily plugged into other graph cut algorithms. Our implementation is publicly available at *http://gridcut.com*.

6. Acknowledgements

We would like to thank all anonymous reviewers for their helpful comments. This work has been supported by the Marie Curie action ERG, No. PERG07-GA-2010-268216 and partially by the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13.

References

[1] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen. Interactive digital photomontage. *ACM Transactions on Graphics*, 23(3):294–302, 2004. 1

[2] C. Arora, S. Banerjee, P. Kalra, and S. N. Maheshwari. An efficient graph cut algorithm for computer vision problems. In *Proceedings of European Conference on Computer Vision*, pages 552–565, 2010. 5

[3] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proceedings of International Conference on Parallel and Distributed Computing Systems*, pages 41–48, 2005. 2

[4] Y. Boykov and M.-P. Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in N-D images. In *Proceedings of International Conference on Computer Vision*, volume 1, pages 105–112, 2001. 1, 4

[5] Y. Boykov and V. Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *Proceedings of International Conference on Computer Vision*, pages 26–33, 2003. 1

[6] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004. 1, 2, 3

[7] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001. 1, 4

[8] B. G. Chandran and D. S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009. 1

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000. 3

[10] A. Delong and Y. Boykov. A scalable graph-cut algorithm for N-D grids. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2008. 2, 3

[11] B. Fishbain, D. S. Hochbaum, and S. Mueller. Competitive analysis of minimum-cut maximum flow algorithms in vision problems. *The Computing Research Repository*, 2010. 1

[12] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. 1

[13] B. Glocker, N. Komodakis, G. Tziritas, N. Navab, and N. Paragios. Dense image registration through MRFs and efficient linear programming. *Medical Image Analysis*, 12(6):731–741, 2008. 1

[14] A. Goldberg. Two-level push-relabel algorithm for the maximum flow problem. In *Proceedings of International Conference on Algorithmic Aspects in Information and Management*, pages 212–225, 2009. 1

[15] A. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988. 1, 2

[16] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. F. Werneck. Maximum flows by incremental breadth-first search. In *Proceedings of Annual European Symposium on Algorithms*, pages 457–468, 2011. 1, 2, 5

[17] A. Hornung and L. Kobbelt. Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information. In *Symposium on Geometry Processing*, pages 41–50, 2006. 1

[18] O. Juan and Y. Y. Boykov. Active graph cuts. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages I: 1023–1029, 2006. 2

[19] O. Juan and Y. Y. Boykov. Capacity scaling for graph cuts in vision. In *Proceedings of IEEE International Conference on Computer Vision*, 2007. 2

[20] P. Kohli and P. H. S. Torr. Dynamic graph cuts and their applications in computer vision. In *Computer Vision: Detection, Recognition and Reconstruction*, pages 51–108. Springer, 2010. 1, 2

[21] V. Kolmogorov. *Graph Based Algorithms for Scene Reconstruction from Two or More Views*. PhD thesis, Cornell University, 2003. 3, 4

[22] V. Lempitsky and Y. Boykov. Global optimization for shape fitting. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2007. 1

[23] V. Lempitsky and D. Ivanov. Seamless mosaicing of image-based texture maps. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2007. 1

[24] Y. Li, J. Sun, and H.-Y. Shum. Video object cut and paste. *ACM Transactions on Graphics*, 24(3):595–600, 2005. 1

[25] J. Liu and J. Sun. Parallel graph-cuts by adaptive bottom-up merging. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 2181–2188, 2010. 1, 2, 3

[26] J. Liu, J. Sun, and H.-Y. Shum. Paint selection. *ACM Transactions on Graphics*, 28(3):69, 2009. 2

[27] H. Lombaert, Y. Y. Sun, L. Grady, and C. Y. Xu. A multilevel banded graph cuts method for fast image segmentation. In *Proceedings of IEEE International Conference on Computer Vision*, pages I: 259–265, 2005. 2

[28] C. Rother, V. Kolmogorov, and A. Blake. Grabcut - Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23(3):309–314, 2004. 1

[29] M. Rubinstein, A. Shamir, and S. Avidan. Improved seam carving for video retargeting. *ACM Transactions on Graphics*, 27(3):16, 2008. 1

[30] A. Shekhovtsov and V. Hlavac. A distributed min-cut/maxflow algorithm combining path augmentation and push-relabel. In *Proceedings of the International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition*, page 14, 2011. 2

[31] P. Strandmark and F. Kahl. Parallel and distributed graph cuts by dual decomposition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 2085–2092, 2010. 2

[32] D. Sýkora, J. Dingliana, and S. Collins. LazyBrush: Flexible painting tool for hand-drawn cartoons. *Computer Graphics Forum*, 28(2):599–608, 2009. 1, 4

[33] R. S. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 30(6):1068–1080, 2008. 4

[34] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Proceedings of Workshop on Visual Computer Vision on GPUs*, 2008. 2, 3

[35] G. Vogiatzis, P. H. S. Torr, and R. Cipolla. Multi-view stereo via volumetric graph-cuts. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 391–398, 2005. 1

Appendix

In this section we present the details of node enumeration and neighbor index computation on the example of 4-connected 2D grid (see supplementary material for other neighborhood systems). In the blocked layout with 8×8 blocks, the index u of a node with grid coordinates x and y is evaluated using the formula

$$u = (x\&7)+((y\&7)\ll 3)+((x\&\sim 7)\ll 3)+W\cdot(y\&\sim 7),$$

where W is width of the padded grid. To compute the indices of a node's neighbors, we make use of the fact that in a 4-connected grid the index of each neighbor can differ from the current node's index by only two possible additive constants, depending on whether the two nodes are in the same or different blocks. In the 8×8 blocked layout, we can decide between the two situations by examining the six least significant bits of the node indices. The bits have a specific pattern on the block's boundary. For instance, the lower three bits are always 000 on the left boundary and the higher three bits are always 111 on the bottom boundary. To compute a neighbor's index, we first check whether the node lies on the block's boundary by comparing the relevant bits and pick one of the constants accordingly. To compute the left, right, top and bottom neighbor of a node with index u, we use the functions

$$\begin{aligned}
left(u) &= u \;\&\; 000111_b \;?\; u-1 \;:\; u-57\\
right(u) &= (\sim u) \;\&\; 000111_b \;?\; u+1 \;:\; u+57\\
top(u) &= u \;\&\; 111000_b \;?\; u-8 \;:\; u-Y_{ofs}\\
bottom(u) &= (\sim u) \;\&\; 111000_b \;?\; u+8 \;:\; u+Y_{ofs},
\end{aligned}$$

where $Y_{ofs} = 8\cdot(W-8+1)$. The use of the ternary operator ? hints the compiler to generate conditional moves instead of branches. This is beneficial as it avoids latencies incurred by branch mispredictions.