Library Documentation

Christian Fischer, Universität Innsbruck

June 23, 2017

Contents

1 General Information

1.1 CSR-Format

All Algorithms which work with Graphs in CSR-Format require the number of edges to be saved as the last entry in the offset array. Concretely a Graph with n Vertices will have an offset array of $n+1$, saving the amount of edges m at offset[m]. Otherwise Algorithms are not guaranteed to terminate correctly.

2 Algorithms

2.1 Breadth First Search

```
void bfs_parallel_baseline(Graph* graph,
                cl_uint* out_cost,
                 cl_uint* out_path,
                  unsigned source,
                   unsigned device_num,
                    unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- out_cost: An array of type cl_uint of length n to store the distance from the source node.Vertices which can't be reached by the Source Node have cost CL_UINT_MAX.
- out_path: An array of type cl_uint of length n to store the predecessor of each node. The source Node will have itself as a predecessor. Vertices which can't be reached by the Source Node have Predecessor CL_UINT_MAX.
- source : The source node for the traversal
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: For every Vertex a Thread is spawned. The Edge List of each Vertex is processed sequentially.

```
void bfs_parallel_workgroup(Graph* graph,
                cl_uint* out_cost,
                 cl_uint* out_path,
                  unsigned source,
                   unsigned device_num,
                    unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- out_cost: An array of type cl_uint of length n to store the distance from the source node.Vertices which can't be reached by the Source Node have cost CL_UINT_MAX.
- out_path: An array of type cl_uint of length n to store the predecessor of each node. The source Node will have itself as a predecessor. Vertices which can't be reached by the Source Node have Predecessor CL_UINT_MAX.
- source : The source node for the traversal
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: Workgroups are processing Edgelists in parallel. Use this if the distribution of outdegrees is highly irregular.

2.2 Single Source Shortest Path

```
void dijkstra_parallel(Graph* graph,
             unsigned source,
              unsigned device_num,
               cl_float* out_cost,
                cl_uint* out_path,
                 unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- source : The source node for the traversal
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- out_cost: An array of type cl_float of length n to store the distance from the source node.Vertices which can't be reached by the Source Node have cost CL_FLOAT_MAX.
- out_path: An array of type cl_uint of length n to store the predecessor of each node. The source Node will have itself as a predecessor. Vertices which can't be reached by the Source Node have Predecessor CL_UINT_MAX.
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: This algorithms works using Atomic Function. It's not guaranteed to work properly.

```
void sssp_normal(Graph* graph,
           unsigned source,
            cl_float* out_cost,
             cl_uint* out_path,
              unsigned device_num,
               unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- source : The source node for the traversal
- out_cost: An array of type cl_float of length n to store the distance from the source node.Vertices which can't be reached by the Source Node have cost CL_FLOAT_MAX.
- out_path: An array of type cl_uint of length n to store the predecessor of each node. The source Node will have itself as a predecessor. Vertices which can't be reached by the Source Node have Predecessor CL_UINT_MAX.
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: This Algorithms avoids global Synchronization using the Edge-Vertex-Message Model.

```
void sssp_opt(Graph* graph,
         unsigned source,
          cl_float* out_cost,
           cl_uint* out_path,
            unsigned device_num,
             unsigned long *time,
              unsigned long *precalc_time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- source : The source node for the traversal
- out_cost: An array of type cl_float of length n to store the distance from the source node.Vertices which can't be reached by the Source Node have cost CL_FLOAT_MAX.
- out_path: An array of type cl_uint of length n to store the predecessor of each node. The source Node will have itself as a predecessor. Vertices which can't be reached by the Source Node have Predecessor CL_UINT_MAX.

- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.
- precalc_time: A pointer to a unsigned variable to store the execution time for the preprocessing

Information: This Algorithms works like sssp_normal, but instead applying lightweight Optimization. Test results show that the preprocessing for the implemented Optimizations take to much time as that they are rentable. If the user of the library can't implement a faster algorithm for the preprocessing it's recommended to use sssp_normal instead.

```
bool bellman_ford(Graph* graph,
           unsigned device_num,
            cl_float* in_cost,
             bool* negative_cycles)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- in_cost: An array of type cl_float of length n which stores the output of a previous sssp algorithms.
- negative_cylces: A pointer to a array of n bool values. At least one vertex which is part of a negative cycle will be marked with true on its position in this array.

Return Value: Returns false if no negative cycle was found, true otherwise.

2.3 All Pair Shortest Path

```
void parallel_floyd_warshall_row(cl_float** in_matrix,
                cl_float** out_matrix,
                 cl_uint** out_path,
                  unsigned length,
                   size_t device_num,
                    unsigned long * time)
```

Parameters:

- in_matrix : A pointer to a 2 Dimensional Array of size $n \times n$, representing the input Graph as Adjacency Matrix
- out_matrix: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the costs
- out_path: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the predecessors

- length: The amount of Vertices in the Graph, i.e. n.
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: Implementation of the APSP Problem. The Kernel works on global memory.

```
void parallel_floyd_warshall_col(cl_float** in_matrix,
                cl_float** out_matrix,
                 cl_uint** out_path,
                  unsigned length,
                   size_t device_num,
                    unsigned long * time)
```

Parameters:

- in_matrix : A pointer to a 2 Dimensional Array of size $n \times n$, representing the input Graph as Adjacency Matrix
- out_matrix: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the costs
- out_path: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the predecessors
- length: The amount of Vertices in the Graph, i.e. n.
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: Implementation of the APSP Problem. The Kernel works on global memory but on the transposition of the Graph. This can improve cache-efficiency, especially on the CPU.

```
void parallel_floyd_warshall_workgroup(cl_float** in_matrix,
                cl_float** out_matrix,
                 cl_uint** out_path,
                  unsigned length,
                   size_t device_num,
                    unsigned long * time)
```

Parameters:

- in_matrix : A pointer to a 2 Dimensional Array of size $n \times n$, representing the input Graph as Adjacency Matrix
- out_matrix: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the costs

- out_path: A pointer to a 2 Dimensional Array of size $n \times n$ for saving the predecessors
- length: The amount of Vertices in the Graph, i.e. n.
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: Implementation of the APSP Problem. The Kernel works on tiles of the input matrix and uses shared memory. This approach should be the fastest of all APSP implementations when applying on GPUs.

2.4 Tranpose

```
Graph* transpose_parallel(Graph* graph,
               size_t device,
               d unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- device: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Return Value : Returns a pointer to a struct Graph representing the Transpose of the input Graph. This variable should explicitly be freed using the Function freeGraph(Graph* graph) when its not in use anymore.

Information: CPU's wont benefit from this parallel Implementation.

2.5 Topological Ordering

```
void topological_order_normal(Graph* graph,
                 cl_uint* out_order_parallel,
                  unsigned device_num,
                   unsigned long *time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- out_order_parallel: A pointer to a array of cl_uint values. Is used for the output , i.e. each Vertex is assigned a Value between 0 and CL_UINT_MAX. Vertices with the same number are equal in the ordering. If a Vertex has the order CL_UINT_MAX, the graph was cyclic, thus an ordering is not possible

- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.

Information: This Algorithms avoids global Synchronization using the Edge-Vertex-Message Model.

```
void topological_order_opt(Graph* graph,
               cl_uint* out_order_parallel,
                unsigned device_num,
                 unsigned long *time,
                  unsigned long *precalc_time)
```

Parameters:

- graph : A pointer to a struct Graph, representing the input Graph in CSR-Format
- out_order_parallel: A pointer to a array of cl_uint values. Is used for the output , i.e. each Vertex is assigned a Value between 0 and CL_UINT_MAX. Vertices with the same number are equal in the ordering. If a Vertex has the order CL_UINT_MAX, the graph was cyclic, thus an ordering is not possible
- device_num: The number of the device on which the Algorithm should be executed. Devices are enumerated across all platforms
- time: A pointer to a unsigned variable to store the execution time excluding the building of the kernel.
- precalc_time: A pointer to a unsigned variable to store the execution time for the preprocessing

Information: This Algorithms works like topological_order_normal, but instead applying lightweight Optimization. Test results show that the preprocessing for the implemented Optimizations take to much time as that they are rentable. If the user of the library can't implement a faster algorithm for the preprocessing it's recommended to use topological_order_normal instead.