Edge coloring bipartite multigraphs for dynamically configuring optical switches

JAN DE NEVE[1,*], ZIYUE ZHANG[1], WOUTER TAVERNIER[1], DIDIER COLLE[1], AND MARIO PICKAVET[1]

[1] Department of Information Technology, Ghent University - imec, Technologiepark-Zwijnaarde 126, Ghent, Belgium
[*] janf.deneve@ugent.be

Compiled June 2, 2025

Multi-chip GPUs interconnected by a photonic network-on-wafer are a promising technology to further increase performance of GPUs. The network control algorithm managing dynamic bandwidth allocation (DBA) in this network needs to execute very frequently so that resources can be optimally used. This algorithm relies on edge coloring bipartite multigraphs to translate inter-chip bandwidth demands into updated routing tables for the GPU chips and optical switches in the network. In this work, we design fast edge coloring algorithms, both approximate and exact, for bipartite multigraphs. These algorithms are tailored to the high edge multiplicities of the multigraphs in this research. The runtimes are optimized by using efficient data structures and introducing pre- and post-processing. These new algorithms are up to 20x faster than the state-of-the-art baseline algorithm. New simulations show that with such low reconfiguration periods, DBA has the potential to double the performance of a high-traffic GPU workload compared to a static network with the same bandwidth.

http://dx.doi.org/10.1364/ao.XX.XXXXXX

1. PHOTONIC NETWORK-ON-WAFER

Developments in research areas like machine learning and data analytics highly depend on increasing computing power of graphics processing units (GPUs). In the past, GPUs kept up with this demand by increasing the number of transistors on a chip. In recent years, transistor downscaling has slackened, forcing the size of a single chip to increase. Since bigger chips are more prone to defects during manufacturing [1], the yield is lower and the fabrication cost of large monolithic GPUs increases.

For this reason, multi-chip GPUs are favored over single-chip GPUs. Multiple pretested chips are integrated and connected on a silicon wafer but they behave as one logical GPU on the software level. Typically, electrical wires are used for interconnecting the different chips on the wafer [2, 3]. This technology works well for connecting multiple cores within one chip as this concerns wire lengths of less than one centimeter. Electrical wires have a major drawback, however, when it comes to connecting different chips on a wafer. The lengths of the inter-chip interconnects on the wafer can be tens of centimeters. Electrical interconnects cannot provide a high bandwidth density at low power consumption over such long distances [4]. This poses a fundamental limitation of multi-chip GPU systems with electrical interconnects.

The photonic network-on-wafer (NoW) GPU architecture aims to overcome this limitation. A GPU chip is grouped with its local memory stack into a so-called GPU tile. Any connections within one GPU tile can be realized with electrical wires, as low power can still be achieved at the sub-centimeter scale. Multiple GPU tiles are then mounted on a wafer and interconnected through a photonic network layer to form the multi-GPU system. When it comes to high bandwidth at low power, the photonic interconnects outperform the electrical wires for three major reasons. First, photonic waveguides in silicon can be spaced more closely than electrical transmission lines, resulting in a higher bandwidth density. Second, wavelength division multiplexing (WDM) can be used to guide multiple optical signals at once, which is not possible with electrical interconnects. Third, optical interconnects can achieve low power consumption even at ranges of tens of centimeters thanks to low-loss waveguide technology [5]. Power consumption of electrical interconnects, on the other hand, scales badly with increasing length. Photonic NoW thus proves to be a promising architecture for future multi-GPU systems. More details on why photonic NoW is beneficial and how this architecture can be realized are found in [6].

The difference between multi-GPU systems with electrical interconnects and multi-GPU systems with photonic interconnects goes beyond the physical components. The different working mechanism of optical switches compared to electrical routers im-

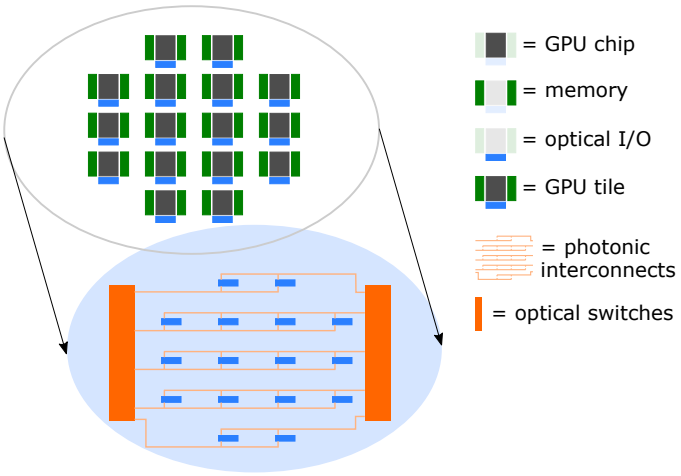

Fig. 1. Main components of the multi-chip GPU on wafer

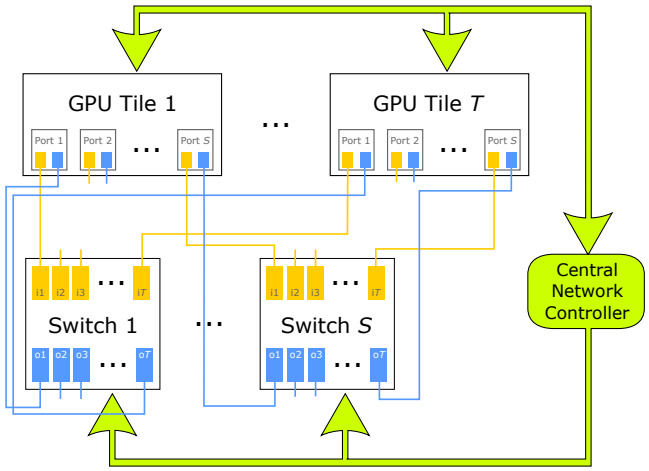

Fig. 2. Interaction between the central controller and the network of GPU tiles and switches

poses a new way of controlling the network. Photonic networks have previously been designed at the chip level for network-on-chip (NoC) architectures. However, such photonic NoC architectures are not the best solution when applying them directly to a photonic NoW system for two reasons. First, the area footprint of an NoW (10^4-10^5mm^2) far exceeds that of an NoC (10^2-10^3mm^2). Such a large area allows for multiple large-footprint photonic devices, like high-radix optical switches, that are unfeasible for an NoC. Second, the bandwidth demands per network node are much higher for an NoW than for an NoC. On the one hand, the nodes of the network in an NoC correspond to a single core or memory channel. The per-node bandwidth of an NoC should therefore only suffice for inter-core and core-memory traffic. On the other hand, nodes in an NoW network represent a GPU chip and memory tile. This results in higher bandwidth requirements for an NoW. Multiple parallel high-radix switches can meet this demand for an NoW. Advantages of high-radix optical switches include high bandwidth per node, low hop count for routing, and simple network topologies. A suitable candidate for such an optical switch in the NoW architecture is a Mach-Zehnder interferometer (MZI). It has been shown that MZI switches can have a high radix, up to 64×64, at nanosecond switching times [7]. An MZI with push-pull microrings can perform wavelength selective routing [8], which allows for a finer granularity when routing the traffic in the network. Figure 1 displays a schematic outline of the components of the multi-chip GPU on wafer.

In this work, we propose fast methods for edge coloring bipartite multigraphs, which are crucial to controlling the network in the multi-GPU system. Section 2 elaborates on the network control algorithm that ensures efficient usage of the network resources. It is also explains why (fast) edge coloring algorithms for bipartite multigraphs are essential to the network control algorithm. Section 3 introduces efficient implementations of sequential coloring algorithms. Different approaches are also discussed to get a solution as close as possible to an optimal solution. Section 4 discusses coloring by directly assigning premade solutions. This method is very fast but only serves as a pre-processing step. Section 5 revisits the *augment* algorithm previously explained by Gabow [9]. We apply it for post-processing suboptimal solutions as well as a full coloring algorithm on its own. Finally, Section 6 compares the runtimes of all proposed methods and analyzes which algorithms perform best for the graphs considered in this research.

2. DYNAMIC NETWORK CONTROL

The photonic NoW could be used to form a static uniform network among the GPU tiles. However, the network can also adapt to the demand patterns among GPU tiles by dynamically reconfiguring the optical switches. The advantages of this dynamic approach have been reported both in the context of a multi-GPU [10] as in similar contexts, like datacenters with a network of optical reconfigurable switches [11], and even for communication networks in general [12]. This method is also called dynamic bandwidth allocation (DBA).

A central controller is responsible for configuring the optical switches and updating the routing tables of the GPU tiles. During a certain time period, the control mechanism consists of two parts. Throughout the entire period, the traffic load between each pair of GPU tiles is observed. This gives an estimation of the traffic load for the next period. Traffic loads usually do not change much in subsequent periods, given that the duration of one period is short enough. At the end of the period, the switches are reconfigured such that the new bandwidth among the GPU tiles reflects their traffic demands as closely as possible. At the same time, the routing tables of the GPU tiles are also updated so that new traffic is sent over the correct channels in the reconfigured switches. Figure 2 shows the direction of control messages (green arrows) between the central controller and the network of GPU tiles and switches. The central controller sends updated routing tables to the GPU tiles and the optical switches, the GPU tiles send the measured traffic demand of one period to the central controller as input for the next reconfiguration round. The period duration is the sum of the runtime of the network control algorithm discussed further on and the time needed to reconfigure the optical switches. Since the network control algorithm is much slower (∼10µs) than the optical switch reconfiguration (∼10ns) [13, 14], it is the main factor in determining the final duration of one period. A runtime of tens to a few hundred microseconds is feasible, which is short enough for many workloads to profit from the DBA [10, 15].

The benefits of DBA come with the variety of traffic demands observed in applications. A static, uniform bandwidth distribution is simpler and has less overhead than dynamic switching, but it only suits quasi-uniform traffic demands or applications with low bandwidth usage. Many applications have a more

complex traffic demand and require the dynamic approach for optimally using the network resources. The increased dynamic bandwidth results in a better overall performance of the GPU system, as off-chip bandwidth is a known bottleneck for many benchmarks [16, 17].

A network control algorithm is needed to translate the monitored traffic demands into a suited configuration of the optical switches. For the algorithm, a system with T GPU tiles and C available unidirectional channels is assumed. The number of available channels C is an abstraction of S physical switches that can operate on W wavelengths independently. This results in $C = SW$ channels for the entire system. One channel corresponds to how one specific wavelength is routed in one specific switch.

In Figure 2, each yellow or blue line represents a waveguide bundling W wavelengths. One waveguide goes from the p^{th} output port of GPU tile T_i to the i^{th} input port of switch S_p. Switch S_p has W different channels, one for each wavelength $\lambda_1 \dots \lambda_W$ that arrives at its inputs. Each wavelength within the switch is independently routed through a different channel. At the j^{th} output port of switch S_p, again the W wavelengths $\lambda_1 \dots \lambda_W$, possibly coming from different input ports of this switch, are bundled into one waveguide. This waveguide finally goes to the p^{th} receiver port of GPU tile T_j.

The input of the network control algorithm is a $T \times T$-matrix $\mathbf{d}$, where $d_{i,j}$ is the number of channels requested for the unidirectional communication from tile i to tile j. $\mathbf{d}$ is called the traffic demand matrix. Evidently, the main diagonal of $\mathbf{d}$ consists of zeros, as no tile needs to communicate with itself.

Any inputs can be considered for the algorithm, but physical implementation and realistic bandwidth requirements make some inputs more relevant to investigate. Considering the typical footprint of GPU tiles compared to the surface area of the wafer, $T = 16$ is the most realistic input size, but other values of T can also be considered. Next to the footprint, the increased heat dissipation difficulty is another reason to avoid adding more GPU tiles. For the number of channels, the most relevant inputs are $C \in [256..512]$. Considering 4 GB/s bandwidth for one channel, i.e. one wavelength in an optical switch, this gives a total bandwidth of 1024 GB/s to 2048 GB/s per tile. No significant improvement in the overall performance is observed for bandwidths exceeding 2048 GB/s [6]. Lower values of C correspond to a system that cannot meet bandwidth requirements for some high-traffic applications. Higher values of C increase the overhead of the network control without improving performance.

A. Network control algorithm in three phases

In [18], we proposed a network control algorithm consisting of three phases. The first two phases can be seen as a pre-processing step to finetune the traffic demand matrix. The actual translation from traffic demand matrix to optical switch configuration takes place in the third phase.

In phase 1, each row and column of $\mathbf{d}$ is rescaled to better fit the available network resources. If a row/column sums up to a value higher than C, the row/column is scaled down until the demand can be met by the available switches. This case cannot be directly measured from the inter-GPU traffic because measured traffic can never exceed the system capacity, but GPUs can still signal that their assigned bandwidth was insufficient during the last period. If the sum of a row/column is very low, this row/column is scaled up to make better use of the available resources. Next to the scaling, it is also made sure that each pair of tiles has at least one channel available. When an entry $d_{i,j}$ of the traffic demand matrix is zero (i.e. no traffic was sent from tile i to tile j during the previous period), this entry is set to one. Each pair of tiles should have a channel available to avoid latencies when these tiles suddenly start communicating. In phase 2, individual entries of the traffic demand matrix are increased until all channels are maximally utilized. When phase 2 is finished, every entry $d_{i,j}$ ranges from 1, the minimum imposed during phase 1, to $\Delta - (T-2)$. This maximum value means that all other entries, except on the main diagonal, of row i and column j are 1 and that $d_{i,j}$ is assigned the highest value allowed such that its row or column do not sum up to a value higher than Δ. In the extreme example where $d_{i,j} = \Delta - (T-2)$, GPU i is transmitting almost exclusively to GPU j and GPU j is receiving almost exclusively from GPU i.

In phase 3, the traffic demand matrix is first turned into a bipartite multigraph with two partitions called U and V. Partition U has T vertices, each representing the transmitters of one GPU tile. Partition V also has T vertices, representing the receivers of the GPU tiles. For every tuple $(i, j) \in [1..T]^2$, $d_{i,j}$ parallel edges are drawn between the i^{th} vertex of partition U and the j^{th} vertex of partition V. A multi-edge with multiplicity $d_{i,j}$ is equivalent to $d_{i,j}$ parallel edges in the multigraph. These notations will be used interchangeably throughout this paper. The resulting multigraph can now be edge colored. An edge coloring of a graph G assigns a color to each edge of G such that no edges incident to the same vertex are given the same color. Each color in the edge coloring of the bipartite multigraph represents a different channel. Edges of color c therefore represent waveguides routed through the c^{th} channel. This c^{th} channel corresponds to the $(c \bmod W)^{th}$ wavelength in the $\lfloor \frac{c}{W} \rfloor^{th}$ physical switch. It is clear that a valid edge coloring results in a valid switch configuration. A transmitting GPU tile cannot send data to two different receiving tiles over the same channel, since a signal arriving in the optical switch cannot be split into two separate signals. This constraint is not violated, since two edges departing from the same vertex of partition U will never have the same color. Similarly, two different transmitting GPU tiles cannot send data to the same receiving GPU tile over the same channel, since two signals arriving in the optical switch cannot be merged into one outgoing signal. This is guaranteed because two edges arriving in the same vertex of partition V will never have the same color. It can be shown that the minimum number of colors required for edge coloring a bipartite multigraph equals the maximum degree Δ of this graph [19]. The graphs in the network control algorithm always satisfy $\Delta = C$ as a result of the rescaling in phase 1 and 2. It is concluded that an exact edge coloring algorithm will therefore yield a valid switch configuration that uses all available channels.

Figure 3 shows the different steps of the algorithm for an example with $T = 4$ and $\Delta = 4$. In phases 1 and 2, the entries of the initial traffic demand matrix (a) are increased. After phases 1 and 2, the traffic demand matrix (b) corresponds to an uncolored, Δ-regular, bipartite multigraph (c). In phase 3, the graph is colored with Δ colors (d). A valid switch configuration can be deduced from this coloring. The value $\Delta = 4$ is equal to the number of channels $C = SW = 4$. For example, this could be a system with two physical switches, each operating on two wavelengths independently, so $S = 2$ and $W = 2$. Each color in the colored graph (d) corresponds to one channel. The color red could correspond to switch S_1 routing wavelength λ_1 as follows: $T_1 \to T_2, T_2 \to T_1, T_3 \to T_4, T_4 \to T_3$.

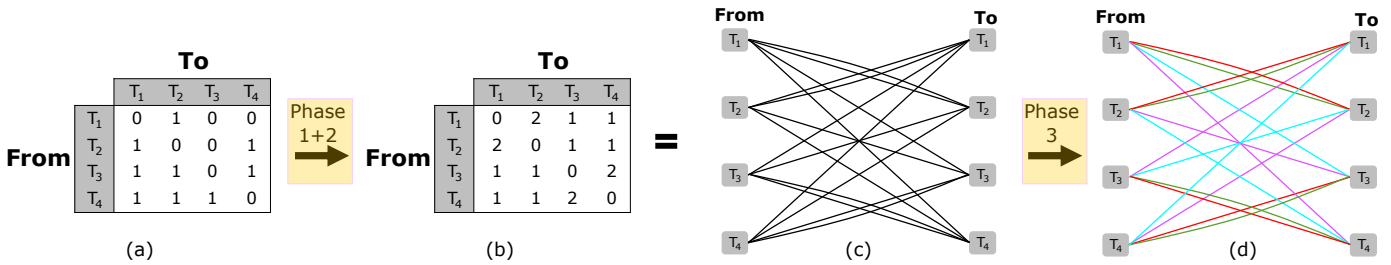

Fig. 3. The network control algorithm starts with a traffic demand matrix and ends with a colored multigraph

B. Existing edge coloring algorithms

Previous authors have studied edge coloring algorithms for bipartite multigraphs. The complexities of these algorithms are expressed as a function of the number of edges E, the maximum degree Δ, and the number of vertices V. Since the number of tiles T is more relevant for this work, V is replaced by T using the equation $V = 2T$. Phases 1 and 2 also make the graphs almost Δ-regular, meaning that $E \approx T\Delta$. This can provide additional insight in the performance of algorithms applied to the specific type of bipartite multigraphs in this research.

Gabow [9] presented multiple algorithms for edge coloring bipartite (multi)graphs. The method *augment* is initially designed to color one edge of a bipartite (multi)graph in $O(T)$ time. The $O(T)$ time stems from up to $2T$ edges that need to be recolored to obtain an available color for the new edge. A color is available at an edge if this color is not yet used by any neighboring edge. Two edges are neighbors or neighboring edges if they share at least one vertex. A full-fledged edge coloring algorithm is obtained when *augment* is sequentially applied to each edge of the graph with total time complexity $O(TE) = O(T^2\Delta)$. A second exact edge coloring algorithm, *color-by-partition*, is also discussed by Gabow. The graph is recursively split into smaller graphs, until each of the smaller graphs can be edge colored with 1 color. Reassembly of the small graphs results in an edge coloring for the initial graph. *color-by-partition* runs in $O((E + T^2)\lg\Delta)$ time.

Cole [20] proposed an edge coloring algorithm with $O(E\lg\Delta)$ time complexity. The difference with Gabow's *color-by-partition* comes from an extra step to make sure that graphs with odd Δ are split into two graphs with maximum degree $\frac{\Delta+1}{2}$ and $\frac{\Delta-1}{2}$ respectively. The improved time complexity of Cole's algorithm over *color-by-partition* does not apply, as the graphs in this research have $E \gg T^2$. Gabow's complexity $O((E + T^2)\lg\Delta)$ thus also boils down to $O(E\lg\Delta)$.

In a previous publication, we implemented Gabow's algorithm for phase 3 of the network control algorithm [18]. We altered the algorithm to optimally exploit the high edge multiplicities in the multigraph, resulting in an improved $O(E + T^2\lg\Delta)$ time bound.

C. Need for fast edge coloring algorithms

It is of most importance that the network can quickly react to changes in the inter-chip traffic demand. For the most relevant inputs $T = 16$ and $C \in [256..512]$, the network control algorithm previously took 8ms to calculate a valid switch configuration. This is fast enough for applications running on the multi-GPU system with steady traffic demands over multiple seconds or minutes. However, some applications experience significant changes in their traffic demands on a sub-second timescale. A faster network control algorithm is needed for these applications, otherwise the overhead induced by the network control might surpass the gains achieved by DBA. In the previous implementation, the network control algorithm spent 95% of the runtime in phase 3, executing the edge coloring algorithm. It is clear that faster edge coloring algorithms will mitigate this bottleneck and improve the runtime of the overall network control algorithm.

A fast network control algorithm is vital for having a short reconfiguration period of the network. Fariborz et al. [10] showed that having a 100µs reconfiguration period can result in high overall performance gain compared to a static network in a multi-GPU. Similar architectures with DBA by reconfiguring optical switches have been studied in an HPC context. Here, performance has been reported to increase significantly from having a shorter reconfiguration period [15, 21].

Figure 4 shows the performance increase obtained by DBA for the GEMM (General Matrix Multiplication) benchmark [22], simulated on a customized version of Accel-Sim [23]. We simulate different setups regarding the total bandwidth capacity of each GPU tile and the reconfiguration time of the network. The reported reconfiguration times also include the 10ns that are needed to reconfigure the switches. During the 10ns that a switch is reconfigured, it can not route any traffic. The simulation assumes a 10ns switching time [7, 8].

An improved performance over a static network configuration is observed both for a reconfiguration period of tens and hundreds of microseconds. The shortest reconfiguration times, represented by the red bars, are based on runtimes of the fastest algorithm we will discuss in this paper. They provide more than 2x speedup over their static counterparts with the same bandwidth, represented by the black bars. Even with reconfiguration periods of ten times longer, represented by the blue bars, the static network architecture is still outperformed by 44% and 42% for the 512GBps and 1024GBps case, respectively. These experiments show that DBA has the potential to speed up workloads in a multi-GPU system, given that there is a network control algorithm fast enough to keep up with these reconfiguration periods. The faster the algorithm, the more the performance can be increased.

The network traffic in the GEMM benchmark gives rise to a traffic demand matrix with diagonal patterns. In a diagonal pattern, each tile has a high traffic load with one or a few neighboring tiles and a low traffic load with all other tiles. This is the ideal setup for DBA to outperform a static architecture, as the allocated channels can dynamically accommodate these diagonal patterns. Other workloads can also have more uniformly distributed traffic demands. In this case, DBA is not expected to increase performance, as a static uniform network is already suited for such tasks.

In the following sections, we propose alternative methods for edge coloring bipartite multigraphs, as encountered in phase 3 of the network control algorithm. The multigraphs encoun-

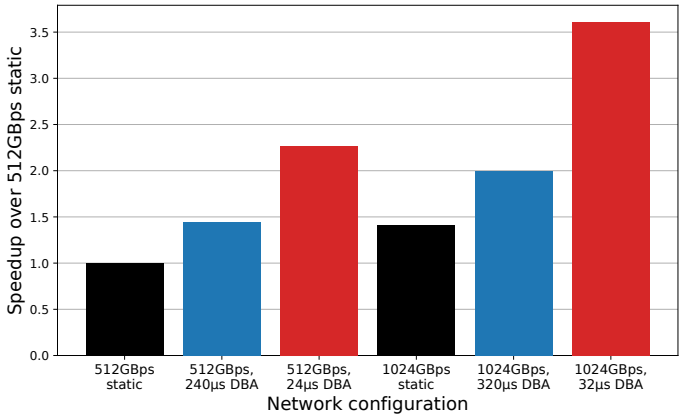

Fig. 4. Impact of DBA on GEMM benchmark ($T = 16$)

tered in this research have very high edge multiplicities, much higher than their number of vertices. This specific property is not discussed in similar literature about edge coloring algorithms for (bipartite) multigraphs. Our work is the first to consider such fast edge coloring algorithms for graphs of this type. We provide both exact and approximate coloring algorithms. An approximate algorithm produces a suboptimal coloring. In a suboptimal coloring, only a part of the E edges of the bipartite multigraph are colored with one of the Δ available colors, the other edges remain uncolored. Since one color corresponds to one channel in a switch, this means that some pairs of tiles are assigned fewer channels than they had initially demanded. Still, these approximate algorithms are worth considering. The traffic demand matrix is only a prediction of future traffic demands, so slight deviations from it may have little impact on the actual performance. Moreover, a suboptimal coloring can still be turned into an optimal one with a post-processing step if an optimal coloring is really needed. In an optimal coloring, all edges of the bipartite multigraph are colored with the Δ available colors and thus all pairs of tiles are assigned the same number of channels as they demanded.

3. SEQUENTIAL COLORING

An easy method for edge coloring a (multi)graph is sequential coloring. It requires only one iteration through all edges, and every edge e is assigned the lowest possible color. This is the color i with the lowest value for which none of the already colored neighbors of e have this color: $i = \min\{x \in \mathbb{N} \mid \forall e' \in N(e) : \text{color}(e') \neq x\}$, where $N(e)$ are the neighboring edges of e and colors are a subset of the natural numbers. Any order of iterating the edges is allowed, but for an efficient implementation in a computer program, all edges forming one multi-edge are colored consecutively.

A. Iteration order

Sequential coloring is not an exact edge coloring algorithm. As a consequence, not all demanded channels will be assigned in the final solution. The number of unassigned channels depends on the iteration order in which the multi-edges are colored. Three iteration orders are considered, each with a different impact on the number of unassigned channels. In the first and simplest approach, the multi-edges are iterated in the same order as they appear in the traffic demand matrix, row major order. We call this order *no priority*. The second approach sorts all multi-edges from large to small multiplicity. This gives a fixed order in which the multi-edges are iterated in the algorithm, called *static priority*. The third approach gives priority to multi-edges with large multiplicity as well, but also takes into account the number of colors still available for every uncolored multi-edge. The priority of a multi-edge is determined by its margin. We define the margin of a multi-edge as the number of available colors at this multi-edge minus the multiplicity of this multi-edge. The algorithm always colors the multi-edge with the smallest margin first. Each time a multi-edge is colored, all margins of the uncolored multi-edges are updated. Since the iteration order is not fixed beforehand, this method is called *dynamic priority*.

Giving priority to multi-edges with a higher multiplicity or smaller margin is beneficial because these multi-edges are "harder" to color. The "easier" multi-edges can later use the leftover colors.

Figure 5 (top) shows the average percentage of color channels that can be successfully assigned by the sequential algorithm for different priorities. Figure 5 (bottom) shows the runtime of a sequential coloring algorithm (see Section 3B) using different priorities. The setup for obtaining these runtimes is discussed in more detail in Section 6.

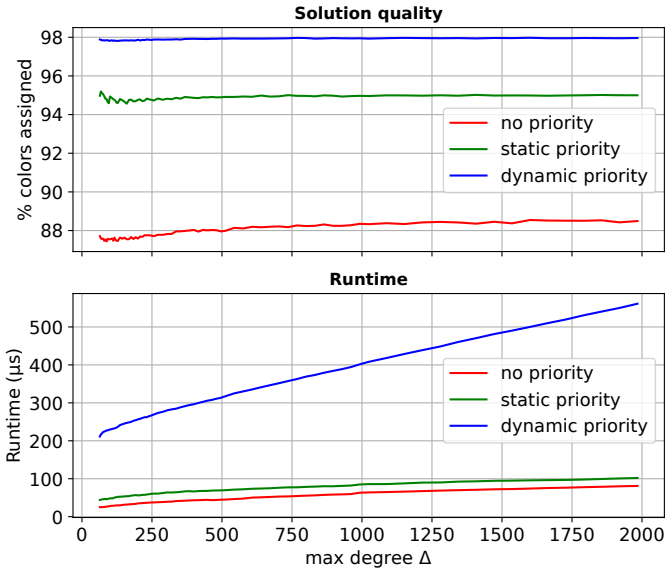

Fig. 5. Sequential coloring with different priorities: better solution quality comes with increased runtime ($T = 16$)

It can be seen that sequential coloring algorithms with *no*, *static*, and *dynamic priority* yield a solution that assigns 88%, 95%, and 98% of channels respectively. The input size has little influence on these numbers. The improved solution does come with an extra runtime cost. *Static priority* takes more than twice as long to execute than *no priority*. The extra cost is due to the sorting of multi-edges before the coloring algorithm starts. *Dynamic priority* is about ten times slower than *no priority*. Every time a multi-edge is colored, the margins of all its uncolored neighbors are updated. Even with an efficiently implemented priority queue, such operations prove to be costly.

Considering the big runtime penalty of using priorities and the fact that *no priority* already provides good solutions, we will

only consider *no priority* in further analyses in this research, unless stated otherwise. A trade-off must always be made between solution quality and runtime. If solutions of better quality are desired, *static priority* can still be considered. *Dynamic priority*, however, will not be part of such trade-offs, as we will later discuss exact coloring algorithms with a lower runtime.

B. Efficient implementation

The outline of the sequential algorithm is simple. For each multi-edge with multiplicity $d_{i,j}$, find the $d_{i,j}$ lowest colors that are not used in any multi-edge with a common vertex, also called a neighboring multi-edge. In a naive implementation, all $2(T-2)$ neighboring multi-edges are checked for the presence of a color c and this is potentially done for all Δ colors. Checking $O(T)$ neighbors for $O(\Delta)$ colors, repeated for all $O(T^2)$ multi-edges leads to a complexity of $O(T^3\Delta)$.

Fortunately, this can be improved. Instead of checking all neighbors for the presence of a color, an array of boolean values can accompany each vertex. The availability of each color is represented by a boolean value and can be looked up in $O(1)$ time. A color is unavailable if it is already used in any of the multi-edges incident to that vertex. Looking up $O(\Delta)$ colors for all $O(T^2)$ multi-edges takes a total $O(T^2\Delta)$ time for the algorithm. Of course, once the $d_{i,j}$ colors of a multi-edge are chosen, the availability must be updated in the vertices to which the multi-edge is incident. This takes $O(d_{i,j})$ time for one multi-edge with multiplicity $d_{i,j}$. Since all multiplicities of multi-edges incident to the same vertex add up to (almost) Δ, $O(\Delta)$ updates are done for coloring all multi-edges of one vertex. This adds up to a $O(T\Delta)$ time complexity for updating all T vertices in the entire algorithm. The color lookups dominate the runtime, so the entire sequential coloring algorithm runs in $O(T^2\Delta)$.

For further improvement of the algorithm, we turn to more efficient data structures. Previously, each vertex was accompanied by an array of boolean values representing the availability of colors. This way, each color must be checked individually, which still takes a long time for graphs with high Δ. We replace the array of booleans with an array of bitmaps. A bitmap is a string of B bits that supports basic operations to be done in parallel on its bits. For best performance, one bitmap should contain as many bits as supported by the processor that will eventually run the program. For most modern processors this will be 64 bits (which can be represented as an unsigned 64-bit integer in programming languages like C++). In general terms, a bitmap with B bits is assumed. In our experiments, $B = 64$.

Very similar to the array of booleans, each bit of the bitmap represents the availability of one color. Generally, $\Delta > B$, so each vertex is accompanied by an array of bitmaps, as one bitmap is insufficient to represent all colors. Though this data structure is similar to the array of booleans, the additional supported operations for bitmaps have a significant impact. Instead of checking one color at a time, B colors can be checked in one operation. This results in $O(1)$ time for lines 8-9,13-14 in Algorithm 1. Without bitmaps, these lines would have a higher time complexity. For each multi-edge with multiplicity $d_{i,j}$, the sequence of available bitmaps is iterated. If the first bitmap has w 1-bits and $w \leq d_{i,j}$, this bitmap is directly assigned to the coloring solution. Now we know for the first B colors whether they are used for this multi-edge, using only few operations. The multiplicity of the multi-edge is then updated, $d_{i,j} \leftarrow d_{i,j} - w$, and the next bitmap is considered. When a bitmap is encountered that has $w > d_{i,j}$, it cannot be fully assigned to the solution. In this case, the shortest prefix of the bitmap with exactly $d_{i,j}$ 1-bits is chosen. The correct prefix length of the final bitmap can be found by a linear search in $O(B)$ time or a logarithmic type search in $O(\lg B)$ time. Now the total number of 1-bits in all bitmaps equals $d_{i,j}$, so the bitmaps contain all the information needed for a valid coloring of this multi-edge. In total, $O(\frac{\Delta}{B})$ bitmaps are checked, all but the last in $O(1)$ time. After coloring a multi-edge, the available colors in the endpoint vertices must be updated. Using the bitmaps, this is again done for B colors at once. The updates therefore take $O(\frac{\Delta}{B})$ time. Adding all complexities gives a $O(\frac{\Delta}{B} + \lg B)$ runtime for coloring one multi-edge. The process is repeated for all $O(T^2)$ multi-edges, giving a total time complexity of $O(T^2(\frac{\Delta}{B} + \lg B))$ for the sequential coloring algorithm using bitmaps of size B. This algorithm is called `sequential-color-bitmap` and is outlined in Algorithm 1.

Algorithm 1. `sequential-color-bitmap`

Require: G is a bipartite multigraph with $2T$ vertices, evenly split over partitions U and V. The multi-edge between the i^{th} vertex of U and the j^{th} vertex of V has multiplicity $d_{i,j}$. The maximum degree is Δ.
Require: Bitmaps of size B are supported.

1: **procedure** SEQUENTIAL-COLOR-BITMAP(G)
2: $\quad \mathbf{b}^U \leftarrow T \times \left\lceil \frac{\Delta}{B} \right\rceil$-matrix of B-sized bitmaps containing only 1-bits
3: $\quad \mathbf{b}^V \leftarrow T \times \left\lceil \frac{\Delta}{B} \right\rceil$-matrix of B-sized bitmaps containing only 1-bits
4: $\quad \mathbf{s} \leftarrow$ empty solution
5: $\quad$ **for** $i = 1 \ldots T$ **do**
6: $\quad\quad$ **for** $j = 1 \ldots T$ **do**
7: $\quad\quad\quad$ **for** $k = 1 \ldots \left\lceil \frac{\Delta}{B} \right\rceil$ **while** $d_{i,j} > 0$ **do**
8: $\quad\quad\quad\quad b \leftarrow \mathbf{b}^U_{i,k}$ & $\mathbf{b}^V_{j,k}$ ▷ & is the bitwise $\wedge$-operator
9: $\quad\quad\quad\quad w \leftarrow$ number of 1-bits in b
10: $\quad\quad\quad\quad$ **if** $w > d_{i,j}$ **then**
11: $\quad\quad\quad\quad\quad b \leftarrow$ smallest prefix of b with $d_{i,j}$ 1-bits
12: $\quad\quad\quad\quad\quad w \leftarrow d_{i,j}$
13: $\quad\quad\quad\quad \mathbf{b}^U_{i,k} \leftarrow \mathbf{b}^U_{i,k}$ & $\neg b$ ▷ $\neg$ is the bitwise inversion operator
14: $\quad\quad\quad\quad \mathbf{b}^V_{j,k} \leftarrow \mathbf{b}^V_{j,k}$ & $\neg b$
15: $\quad\quad\quad\quad \mathbf{s}_{i,j,k} \leftarrow b$
16: $\quad\quad\quad\quad d_{i,j} \leftarrow d_{i,j} - w$
17: $\quad$ **return s**

Figure 6 shows the runtime of `sequential-color-bitmap` and the runtime of the same algorithm without bitmaps. The relative difference between `sequential-color-bitmap` and its counterpart without bitmaps increases with bigger input sizes. This is because the relative contribution of the control overhead in the algorithm decreases. Part of this overhead is the same, regardless of the use of bitmaps. For high input sizes, `sequential-color-bitmap` is about 10 times faster. This is still less than the theoretical speedup of $B = 64$. Note that using bitmaps does not make every step B times more efficient. For example, the array of booleans representing the availability of colors in a multi-edge, can be stored in memory using only one bit per boolean. In that case, the memory usage is as efficient as for `sequential-color-bitmap` and as many memory operations will be needed.

Even for low input sizes, the speed more than doubles, show-

ing the strength of the implementation with bitmaps. Given the clear performance gap, only `sequential-color-bitmap` will be considered as a sequential algorithm in later analyses.

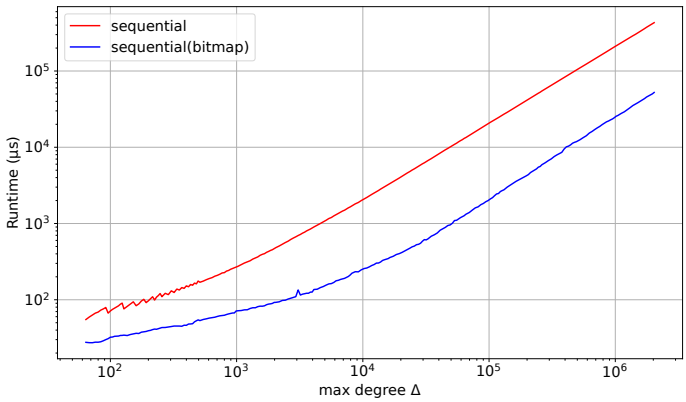

Fig. 6. Speedup of implementation with bitmaps ($T = 16$)

4. DIRECT ASSIGNMENT

The fastest possible way to obtain a coloring is to have a solution ready and only having to check if it is valid. If a (partial) solution is validated, it is directly assigned. Clearly, complete solutions cannot be ready beforehand, as there are infinitely many demand matrices that all have different solutions. For that reason, only partial solutions consisting of one perfect matching are considered. A perfect matching is a set of edges that covers every vertex of the graph exactly once. Hence the edges of this partial solution can be colored with a single color. Still $\left\lfloor \frac{T!}{e} + \frac{1}{2} \right\rfloor$ possible partial solutions remain. This equals the number of permutations of T tiles where no tile is mapped on itself. In graph theory, this is the number of perfect matchings in a crown graph with $2T$ vertices, which is the underlying simple graph of the bipartite multigraphs in this research. A crown graph is a complete bipartite graph with only the horizontal edges missing. For example, if each multi-edge in Figure 3c is merged into one simple edge, it becomes a crown graph with 2×4 vertices. It takes too long to check all these partial solutions with $T = 16$, so a subset is taken for this direct assignment step.

Looking beyond the algorithm itself, some partial solutions can be more interesting to consider than others. For one, some traffic demands might occur frequently in applications running on the multi-GPU system. This indicates which partial solutions have a higher chance of being present in the graph of the edge coloring problem. For another, the optical switches in the system might perform better under certain configurations. Using partial solutions that correspond to such configurations, will force the coloring solution to optimally conform to the physical properties of the multi-GPU architecture. For the experiments in this research, we use randomly generated partial solutions. We put an extra restriction that each set of $T - 1$ subsequent partial solutions forms a full mesh in the multigraph. This restriction makes the partial solutions very well suited for uniform traffic demands.

We now use a set of pre-made partial solutions to simplify the coloring of a graph. If every edge of a partial solution is present in the graph, we remove these edges from the graph and assign one color to them in the final solution. Because the partial solution is a perfect matching, the remaining graph now has degree $\Delta - 1$ and can therefore be $(\Delta - 1)$-colored. This process is repeated until all pre-made solutions are tested. If all edges of a partial solution occur at least k times in the multigraph, the partial solution can be added k times to the final solution at once. This shows the strength of the direct assignment method for multigraphs with high multiplicities.

Due to the limited number of pre-made solutions, many edges of the graph remain uncolored. As opposed to the sequential coloring algorithm, the solution obtained from direct assignment is not close enough to an optimal solution to accept this as a valid switch configuration. Less than half of the demanded channels are assigned when checking a realistic number of pre-made solutions. The direct assignment method is therefore always followed by another algorithm to (partially) color the remaining channels. Direct assignment can thus only be used as a pre-processing step.

Direct assignment has time complexity $O(PT)$ when P different pre-made solutions are checked, each with T edges. The actual value of P should be chosen by comparing the runtime gained from making the graph smaller and the extra cost of checking more partial solutions. Typically, at least $T - 1$ pre-made solutions should be checked as those can form a full mesh in the graph, i.e. together they contain all the edges of the underlying simple graph. Such a uniform pattern occurs frequently in applications and phase 1 of the network control algorithm specifically inserts this pattern in the traffic demand matrix.

5. AUGMENTATION

A weakness of sequential coloring and direct assignment is that obtained colorings are not optimal. The goal is to find a post-processing mechanism that can turn these suboptimal colorings into optimal ones while increasing the total runtime as little as possible. For this, we revisit the *augment* algorithm previously described by Gabow [9]. An edge that is left uncolored by an approximate algorithm can now be colored individually by the *augment* algorithm. The `color-by-augmentation` procedure repeats this process for all uncolored edges in the graph. When an empty solution is taken as the initial solution, the post-processing becomes a full-fledged coloring algorithm.

Algorithm 2 shows the outline of how the *augment* procedure from Gabow is repeatedly used to obtain an exact coloring of a bipartite multigraph. We call this algorithm `color-by-augmentation`. The algorithm is valid for both post-processing and starting from an empty solution. If the algorithm is used as a post-processing step, we first decrease the entry $d_{i,j}$ of the demand matrix by one for every edge between the i^{th} vertex and the j^{th} vertex that is already colored in the suboptimal coloring, and then Algorithm 2 is executed with the updated multiplicities $d_{i,j}$.

The runtime complexity of `color-by-augmentation` is obtained from the algorithm outline. The inside of the while-loop is executed exactly E^* times throughout the algorithm. $E^* \leq E$ is the number of uncolored edges at the start of the post-processing. Lines 9-10 find and change a path of length $O(T)$. This contributes $O(TE^*)$ to the runtime in the worst case. If naively implemented, lines 7-8 take $O(\Delta)$ time to find an absent color. This would make these lines contribute $O(\Delta E^*)$ to the runtime. A better implementation initializes the variables α and β outside the while-loop, because an absent color will never be lower than the previous absent color for the same multi-edge. This way, finding values for α and β will only take $O(\min[T^2, E^*]\Delta)$ time through-

Algorithm 2. `color-by-augmentation`

Require: G is a bipartite multigraph with $2T$ vertices, evenly split over partitions U and V. The multi-edge between the i^{th} vertex of U and the j^{th} vertex of V has (updated) multiplicity $d_{i,j}$.

procedure COLOR-BY-AUGMENTATION(G)
  for $i = 1 \dots T$ **do**
    $u \leftarrow i^{th}$ vertex in partition U
    for $j = 1 \dots T$ **do**
      $v \leftarrow j^{th}$ vertex in partition V
      while $d_{i,j} > 0$ **do**
        α is lowest color absent from u
        β is lowest color absent from v
        Find longest path P starting in v with only α- and β-colored edges
        Swap the colors α and β for all edges in P
        Add α-colored edge between u and v
        $d_{i,j} \leftarrow d_{i,j} - 1$

out the entire algorithm. This brings the total time complexity of `color-by-augmentation` as a post-processing mechanism to $O(TE^* + \min[T^2, E^*]\Delta)$. This boils down to $O(\min[T^2, E^*]\Delta)$ because $E^* \leq T\Delta$ and $T < \Delta$. If `color-by-augmentation` is used as coloring algorithm on itself, the complexity remains $O(T^2\Delta)$. Note that the theoretical bound of the post-processing is only better than the full algorithm if $E^* \ll T^2$. Figure 5 indicates that this can be the case if a sequential coloring is done as a first step.

6. COMPARING RUNTIMES

In this section, runtimes of all algorithms are compared. The inputs of the experiments are multigraphs with $T = 16$ and maximum degrees $\Delta \in [64..2048]$. Each reported runtime in the figures is the average value over 1000 generated input multigraphs. This way, the reported runtimes give a very reliable image of the average performance of the algorithms. The demand matrices of the input multigraphs are randomly generated. Each entry (i, j) is drawn from a uniform distribution. This value corresponds to the traffic between tile i and tile j in a multi-GPU system. We then transform this demand matrix according to phase one and phase two of the network control algorithm described in Section 2. This way, the multigraphs colored in the experiments are representative to those that would occur in the multi-GPU system described in this research.

The algorithms are implemented in C++ and compiled with the g++ compiler using the -Ofast optimization parameter. The experiments are conducted on an AMD EPYC 7773X (2.2 GHz) processor. Note that in a multi-chip GPU environment, it is also possible to run the network control algorithm on dedicated hardware rather than on a CPU. This would push the runtimes of the edge coloring algorithms further down, even to a sub-microsecond scale. The sequential coloring algorithm is particularly suited for this.

The new algorithms are compared to Gabow's algorithm *color-by-partition*, the color algorithm used in [18]. To this end, we made a new implementation of Gabow's algorithm. In this new implementation, the performance is optimized by using bitmaps, a highly efficient data structure that we also used to optimize the sequential coloring algorithm in `sequential-color-bitmap` (see Section 3B). It is most fair to use the same efficient data structures in this implementation as we have used in the implementation of the new coloring algorithms. This way, any difference in performance is inherent to the algorithms and cannot be attributed to a difference in used data structures and programming techniques.

Figure 7 shows the runtime of `sequential-color-bitmap` (red), `color-by-augmentation` (pink), those same two algorithms preceded by `direct-assignment` (blue and cyan respectively), and the optimized version of Gabow's algorithm (orange). The logarithmic axes allow a detailed overview, even with the big differences in runtimes.

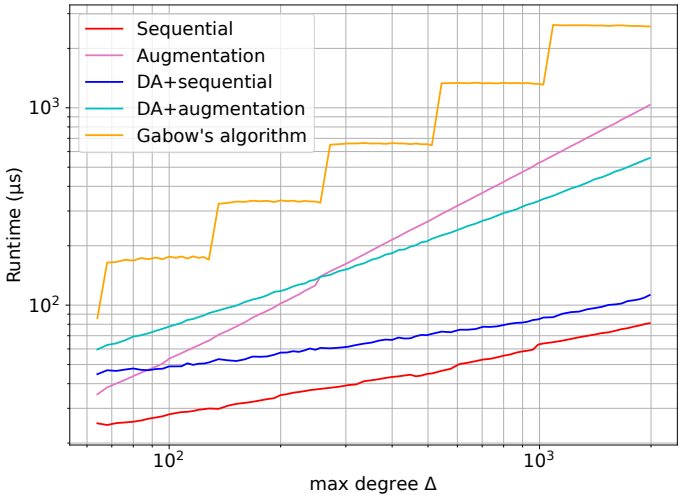

Fig. 7. Runtimes of edge coloring algorithms ($T = 16$)

It can be seen that all algorithms execute in less than 1ms in the most relevant region ($\Delta \in [256..512]$). Gabow's algorithm is the slowest with an execution time of 650µs, despite the improved implementation and the best theoretical complexity. This shows that theoretical complexity is not always a good measure for algorithm performance with realistic inputs. The runtime mostly depends on the depth of the recursion tree ($\lceil \lg \Delta \rceil$) rather than the actual maximum degree Δ. This gives a near constant runtime for input sizes ranging between subsequent powers of two, resulting in the staircase effect in the figure.

`color-by-augmentation` has an execution time of 140µs to 270µs. It is remarkable that it is faster than Gabow's algorithm, as `color-by-augmentation` is based on *augment*, a subprocedure of Gabow's algorithm. Performing the `direct-assigment` preprocessing step before `color-by-augmentation` improves the runtime to around 140µs to 220µs. For lower input sizes, the improvement is canceled out by the extra overhead of executing two separate algorithms.

`sequential-color-bitmap` has the lowest execution time, about 35µs to 45µs. This is almost 20x faster than our baseline, Gabow's algorithm. The `direct-assigment` preprocessing step has no positive impact on the runtime, because both algorithms are so fast that their constant overhead is too significant at this input size. The relative difference between `sequential-color-bitmap` with and without the `direct-assigment` pre-processing step decreases for larger inputs.

In Section 3, we already looked at how well `sequential-color-bitmap` can approximate an optimal coloring. When `sequential-color-bitmap` is preceded by `direct-assignment`, the quality of this approximate solution further increases. Figure 8 shows the solution quality of

`sequential-color-bitmap` with (green) and without (red) the `direct-assignment` pre-processing step. It can be seen that with pre-processing, the solution quality is clearly better than the 88% obtained without pre-processing and goes up to 99% and even higher for input graphs with maximum degree $\Delta = 256..512$. This high solution quality can be explained by the structure of the input graphs. The input graphs always contain some copies of a crown graph as a subgraph. When Δ gets higher, the fraction of all edges covered by these crown subgraphs increases, up to 50%. `direct-assignment` is suited to optimally color such crown subgraphs. The remaining edges are then (suboptimally) colored with `sequential-color-bitmap`, resulting in a final solution of high quality, even more so for graphs with high Δ.

Although the solution quality gets close to 100% for high Δ, it never becomes an optimal solution. `sequential-color-bitmap` therefore remains an approximate algorithm, even with pre-processing. It should also be noted that when a different type of input graph is considered, for example a different value of T or a different structure of the demand matrix, the fraction of the edges covered by crown subgraphs can be a lot lower. In that case, the solution quality of `sequential-color-bitmap` with `direct-assignment` pre-processing is not as close to the optimal, though it is always higher than without the pre-processing.

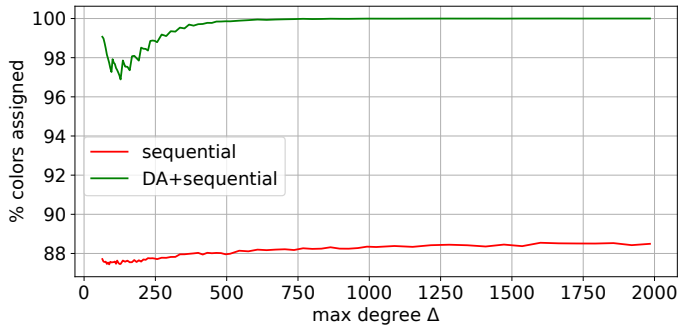

Fig. 8. Solution quality of `sequential-color-bitmap` with and without the `direct-assignment` pre-processing step ($T = 16$)

Regarding approximate algorithms, sequential coloring remains the fastest solution. Regarding exact algorithms, the combination of `sequential-color-bitmap` followed by `color-by-augmentation` runs faster than `color-by-augmentation` alone and both yield an exact solution. `sequential-color-bitmap` can color the majority of edges in a very short time. The remaining channels are colored by `color-by-augmentation` such that an exact coloring is obtained. This approach optimally exploits the advantages of both methods: the speed of `sequential-color-bitmap` and the flexibility and exactness of `color-by-augmentation`. Figure 9 shows the runtimes for the combined algorithm as well as its constituent parts on their own.

In previous runtime analyses, only the *no priority* variant of the sequential coloring algorithm was considered due to its fast execution time. When an exact coloring is obtained by using `sequential-color-bitmap` followed by `color-by-augmentation` as post-processing, the *static priority* variant must be revisited. *Static priority* can already color more channels than *no priority*, leaving less work to be done by the (slower) post-processing. The higher runtime of the *static priority* variant of the sequential algorithm is thus compensated by the reduced input size for the post-processing. Figure 10 shows how the *static priority* variant becomes the best option for input sizes $\Delta > 500$ and $\Delta > 1000$, for the combined algorithms without and with `direct-assigment` as pre-processing step, respectively.

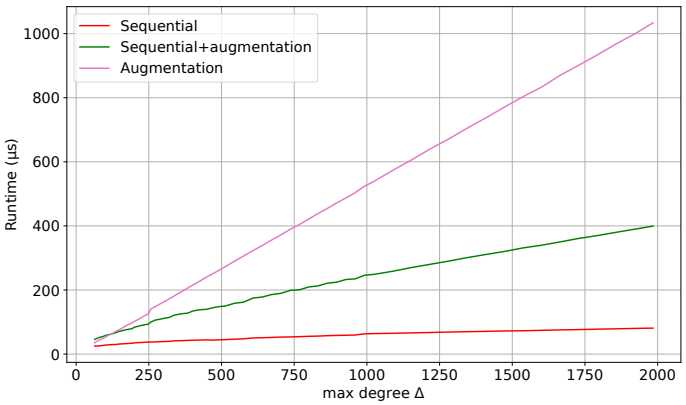

Fig. 9. Augmentation as a post-processing step ($T = 16$)

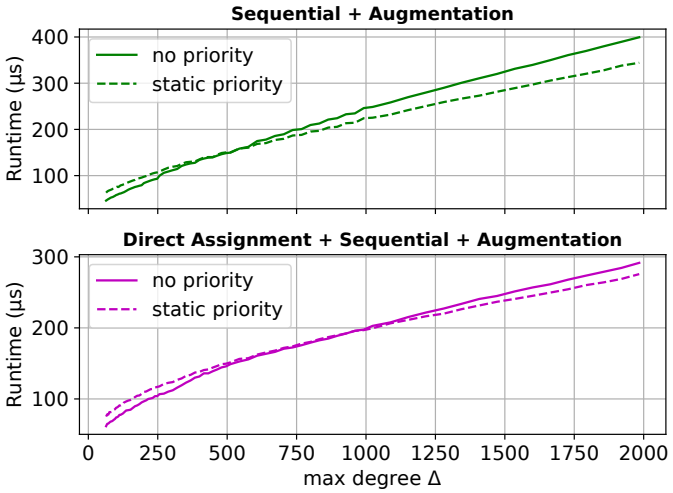

Fig. 10. Using static priority becomes beneficial for the sequential+augmentation combination for higher Δ ($T = 16$)

As stated in Section 2, we also consider values of T other than 16. Figures 11 and 12 show the runtimes of `sequential-color-bitmap` and `color-by-augmentation` for a configuration with $T = 8$, 16, 32, and 64. Both algorithms have a theoretical time complexity that is proportional to T^2. T is doubled for each different configuration in the experiments, so the runtime is expected to quadruple between two subsequent configurations. The solution quality of `sequential-color-bitmap` for the different values of T is again around 88%, not changing significantly for increasing Δ.

For the sequential coloring algorithm, the observed runtimes closely follow the expected quadrupling trend. This confirms that the theoretical complexity closely resembles the actual operations executed in the algorithm. For the augmentation algorithm, the runtimes between subsequent configurations differ a factor of about 3 to 3.5. So in practice, `color-by-augmentation`

scales slightly less than quadratic with T. This can be explained by lines 9 and 10 in Algorithm 2. These lines find and alter a path of maximal length $2T$, but this path can be shorter. At the start of the algorithm, for example, this path will have length 0. Paths with a length close to the maximum will generally only be reached at the end of the algorithm. Since the theoretical upper bound of $O(T)$ is less tight, the deviation from the expected quadrupling is bigger for this algorithm.

Comparing different values of T for the same input size shows the quadratic dependence of the runtimes on parameter T, i.e. the number of GPU tiles. It must be taken into account that for a hypothetical system with more GPU tiles (e.g. $T = 64$), the traffic per node will likely increase. In that case, higher input values should be considered.

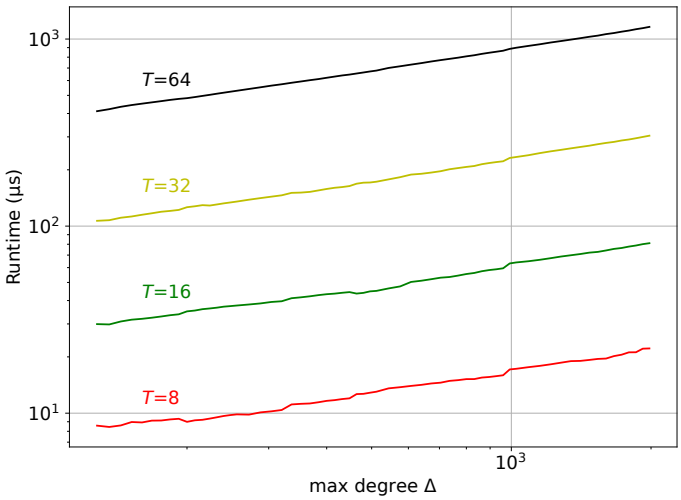

Fig. 11. Runtimes of `sequential-color-bitmap` for different T

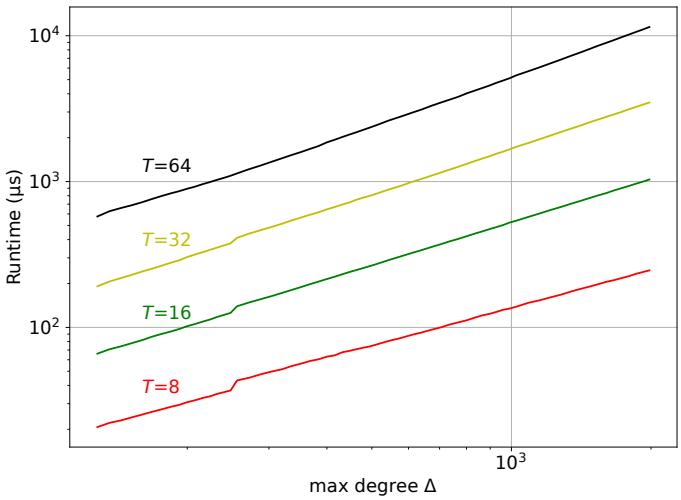

Fig. 12. Runtimes of `color-by-augmentation` for different T

7. CONCLUSIONS

Photonic network-on-wafer is a promising technology for interconnecting GPUs with low delays and energy consumption. Dynamic bandwidth allocation (DBA) can help to further increase bandwidth capacities of the system, but this requires a fast network control algorithm and consequently a fast edge coloring algorithm for bipartite multigraphs. The faster the algorithm, the more beneficial DBA becomes for the performance of the multi-GPU system. We show this added value of DBA with low reconfiguration periods in new simulations, where we used the runtime of the algorithms discussed in this research as parameter.

The unique challenge about coloring the multigraphs encountered in the photonic multi-GPU application is that they have very high edge multiplicities. We optimize our edge coloring algorithms such that they optimally suit this property of the graphs and we make use of specific data structures such that the runtime is reduced to a minimum. This is necessary for use in high-performance applications such as the photonic multi-GPU network. The specific nature of the graphs in this research, with their high edge multiplicities, and how we target this with our tailored algorithms, adds to the relevance and novelty of this research.

We designed different edge coloring algorithms, both exact and approximate, focusing on optimal runtime performance. For the approximate algorithms, a trade-off must be made between runtime and the fraction of channels assigned by the algorithm. `sequential-color-bitmap` with *no priority* has a low runtime and its solution quality of 88% of colors assigned is decent. This algorithm is therefore a suited candidate, certainly for use in workloads with very quickly changing traffic patterns. If traffic demands change more slowly, the workloads may benefit more from a slower algorithm with a higher solution quality. The combination of `direct-assignment` with `sequential-color-bitmap` is a good alternative when a better solution quality is required without too much increase in runtime. An exact algorithm can be considered when the solution quality is the most important factor.

For the exact algorithms, the best approach is to combine all proposed algorithms and make optimal use of each of their strengths. For the most relevant inputs, $T = 16$ and $\Delta \in [256..512]$, the fastest exact algorithm uses `direct-assignment` as a pre-processing step, followed by `sequential-color-bitmap`, and `color-by-augmentation` as post-processing.

Until now, phase 3 of the network control algorithm was a bottleneck, taking up to 95% of the total runtime. Now that we have drastically improved the edge coloring algorithms for phase 3, the bottleneck has shifted to phase 2. It is vital to speed up this phase as well, to get a complete network control algorithm suited for DBA in a multi-GPU system. We are confident that significant improvements can be made, either by altering phase 2 or by integrating phases 2 and 3 to introduce further combined optimizations.

FUNDING

Special Research Fund Ghent University (BOF23-DOC-146, BOF21-GOA-014); Flemish Research Foundation (1S11323N).

DISCLOSURES

The authors declare no conflicts of interest.

REFERENCES

1. C. Stapper, "On murphy's yield integral (ic manufacture)," IEEE Transactions on Semicond. Manuf. **4**, 294–297 (1991).
2. D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," IEEE Micro **37**, 7–17 (2017).
3. D. D. Sharma, "Pci express® 6.0 specification at 64.0 gt/s with pam-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect," 2020 IEEE Symp. on High-Performance Interconnects (HOTI) pp. 1–8 (2020).
4. S. Pasricha and M. Nikdast, "A survey of silicon photonics for energy-efficient manycore computing," IEEE Des. & Test **37**, 60–81 (2020).
5. B. Haq, S. Kumari, K. Van Gasse, J. Zhang, A. Gocalinska, E. Pelucchi, B. Corbett, and G. Roelkens, "Micro-transfer-printed iii-v-on-silicon c-band semiconductor optical amplifiers," Laser & Photonics Rev. **14**, 1900364 (2020).
6. S. Zhang, Z. Zhang, M. Naderan-Tahan, H. SeyyedAghaei, X. Wang, H. Li, S. Qin, D. Colle, G. Torfs, M. Pickavet, J. Bauwelinck, G. Roelkens, and L. Eeckhout, "Photonic Network-on-Wafer for Multichiplet GPUs," IEEE Micro **43**, 86–95 (2023).
7. T. Chu, L. Qiao, W. Tang, D. Guo, and W. Wu, "Fast, high-radix silicon photonic switches," in *2018 Optical Fiber Communications Conference and Exposition (OFC),* (2018), pp. 1–3.
8. Y. Huang, Q. Cheng, A. Rizzo, and K. Bergman, "Push—pull microring-assisted space-and-wavelength selective switch," Opt. Lett. **45**, 2696 (2020).
9. H. N. Gabow and O. Kariv, "Algorithms for Edge Coloring Bipartite Graphs and Multigraphs," SIAM J. on Comput. **11**, 117–129 (1982).
10. M. Fariborz, X. Xiao, P. Fotouhi, R. Proietti, and S. J. B. Yoo, "Silicon Photonic Flex-LIONS for Reconfigurable Multi-GPU Systems," J. Light. Technol. **39**, 1212–1220 (2021).
11. C. A. Caldeira, O. A. De O. Souza, O. Goussevskaia, and S. Schmid, "OpticNet: Self-Adjusting Networks for ToR-Matching-ToR Optical Switching Architectures," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications,* (IEEE, New York City, NY, USA, 2023), pp. 1–10.
12. O. Peres and C. Avin, "Distributed Demand-aware Network Design using Bounded Square Root of Graphs," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications,* (IEEE, New York City, NY, USA, 2023), pp. 1–10.
13. G. Coudyzer, P. Ossieur, J. Bauwelinck, and X. Yin, "A 25gbaud pam-4 linear burst-mode receiver with analog gain- and offset control in 0.25μm sige:c bicmos," IEEE J. Solid-State Circuits **55**, 2206–2218 (2020).
14. X. Wang, A. Vandierendonck, B. Govaerts, T. Pannier, W. Geeroms, C. Meysmans, J. Bauwelinck, and G. Torfs, "A duty-cycle switching 30-gb/s burst-mode cdr with 1.6-ns locking time in 28-nm cmos," IEEE J. Solid-State Circuits pp. 1–13 (2025).
15. M. Y. Teh, Z. Wu, M. Glick, S. Rumley, M. Ghobadi, and K. Bergman, "Performance trade-offs in reconfigurable networks for HPC," J. Opt. Commun. Netw. **14**, 454 (2022).
16. A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture,* (ACM, Toronto ON Canada, 2017), pp. 320–332.
17. A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry, "Mitigating the Memory Bottleneck With Approximate Load Value Prediction," IEEE Des. & Test **33**, 32–42 (2016).
18. Z. Zhang, D. Colle, W. Tavernier, and M. Pickavet, "On the network design and control of an optical network: interconnecting multiple chips on a wafer," J. Opt. Commun. Netw. **15**, 119–132 (2023).
19. C. Berge, *Graphs and Hypergraphs*, Graphs and Hypergraphs (North-Holland Publishing Company, 1973).
20. R. Cole, K. Ost, and S. Schirra, "Edge-Coloring Bipartite Multigraphs in O (E log D) Time," Combinatorica. **21**, 5–12 (2001).
21. M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi, "SiP-ML: high-bandwidth optical network interconnects for machine learning training," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference,* (ACM, Virtual Event USA, 2021), pp. 657–675.
22. "Gemm benchmark," .
23. M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA),* (2020), pp. 473–486.