Global optimization of operand transfer fusion in heterogeneous computing

Christoph Kessler
Linköping University
christoph.kessler@liu.se

ABSTRACT

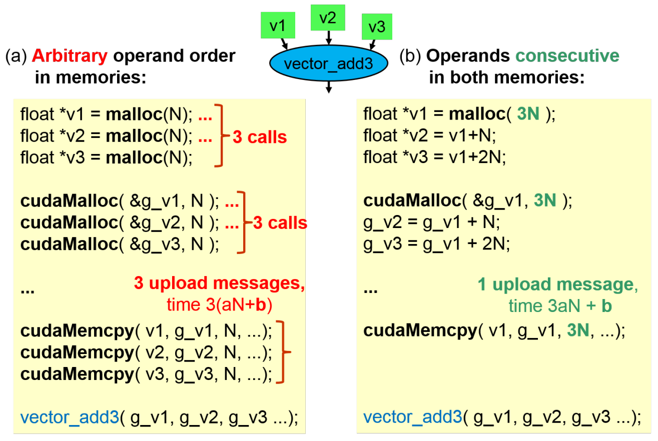
We consider the problem of minimizing, for a dataflow graph of kernel calls, the overall number of operand data transfers, and thus, the accumulated transfer startup overhead, in heterogeneous systems with non-shared memory. Our approach analyzes the kernel-operand dependence graph and reorders the operand arrays in memory such that transfers and memory allocations of multiple operands adjacent in memory can be merged, saving transfer startup costs and memory allocation overheads.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Communications management**; *Compilers.*

KEYWORDS

Heterogeneous computing, data transfer fusion, allocation fusion, GPU, distributed memory, program optimization, kernel dataflow graph, memory mapping, Hamiltonian path


v1
v2
v3
vector_add3
(a) Arbitrary operand order in memories:
(b) Operands consecutive in both memories:
3 calls
3 calls
3 upload messages, time 3(aN+b)
1 upload message, time 3aN + b

```
float *v1 = malloc(N); ...
float *v2 = malloc(N); ...
float *v3 = malloc(N);

cudaMalloc( &g_v1, N ); ...
cudaMalloc( &g_v2, N ); ...
cudaMalloc( &g_v3, N );

...

cudaMemcpy( v1, g_v1, N, ...);
cudaMemcpy( v2, g_v2, N, ...);
cudaMemcpy( v3, g_v3, N, ...);

vector_add3( g_v1, g_v2, g_v3 ...);
```

```
float *v1 = malloc( 3N );
float *v2 = v1+N;
float *v3 = v1+2N;

cudaMalloc( &g_v1, 3N );
g_v2 = g_v1 + N;
g_v3 = g_v1 + 2N;

...

cudaMemcpy( v1, g_v1, 3N, ...);

vector_add3( g_v1, g_v2, g_v3 ...);
```

Figure 1: Local transfer fusion for one call to a ternary vector-add kernel. Naive code generation (case (a)) allocates and transfers each operand separately. If we know (or permute the data layout) that the operands are consecutive in both source and destination memory (case (*b*)), operand transfers and also memory allocations can be merged [8]. This work extend the idea to static *global* transfer fusion for multi-kernel scenarios.

1 INTRODUCTION

With Moore's Law running out of steam, heterogeneous computing will, together with better optimized software, become even more significant for exploiting the remaining performance reserves of CMOS-based hardware technology. Heterogeneous systems combine general-purpose CPUs with GPUs, FPGAs and further types of accelerators designed for special kinds of computations. It is also likely that future non-CMOS-based computing technologies will first become available commercially as accelerators to conventional computers, much as we use discrete GPUs and FPGAs today. Many heterogeneous systems, such as most systems with a discrete GPU, expose a distributed memory structure with separate accelerator device memory such that operand data needs be transferred (more or less) explicitly between main memory and device memory.

Executions of performance-hungry applications such as deep learning are often structured in the form of (sometimes quite complex) *kernel data flow graphs*, i.e., dependence graphs over kernel calls, where the kernel calls might execute either on the CPU or on the accelerator, depending, e.g., on kernel type and operand size. The graph structure can be leveraged for global program optimizations, both statically in compilers and dynamically in runtime systems for heterogeneous systems. For example, the skeleton programming framework *SkePU* allows for global tiling of *lineages* (i.e., acyclic kernel-vector graphs) of skeleton-based kernels at runtime to improve cache hit rates [3].

In this paper, we leverage the global data flow information for an optimization of the necessary array operand data transfers between main memory and device memory, by minimizing the number of transfer messages and, thus, of accumulated transfer startup times. The main idea is based on reordering operand arrays in memory to improve the opportunities for transfer fusion. We thereby generalize over the on-line transfer fusion technique by Li et al. [8] (see also Figure 1 for an example) which is only applicable to a *single* kernel call and ignores other, non-local constraints on the operands' memory mapping.

In this work we make the following contributions:

- We formalize the global static optimization problem of finding a memory mapping for the program's operands such that the number of transfer fusions is maximized, as a graph-based optimization problem, and solve it using heuristic techniques.
- We provide a generic proof-of-concept prototype implementation of our approach and show experimentally that it is effective in eliminating transfer startups and that it has rather low optimization time.
- While the proposed optimization method is not specific to GPUs as accelerators, our prototype includes a source code emitter that generates CUDA source code from a given dataflow graph and the calculated memory mapping. We perform an experimental evaluation on a system with a CUDA GPU, demonstrating a potential for significant speedups for transfer-dominated programs with small to medium-sized array operands.

The remainder of this paper is organized as follows: We begin by motivating our work in Section 2, which includes an experimental quantitative analysis of the impact of transfer fusion and memory allocation fusion on a concrete GPU system. Section 3 formalizes the problem and Section 4 presents the optimization approach. Experimental results follow in Section 5. Section 6 discusses related work, and Section 7 concludes and identifies future work.

2 PERFORMANCE IMPACT OF TRANSFER AND ALLOCATION FUSION

On most architectures with a physically distributed memory, it is reasonable to assume that a data transfer of a block of n data elements takes time $an+b$, with positive machine-specific constants $a, b > 0$, where the transfer startup time b is usually much larger than the per-element transfer time a. Even for the same machine, the coefficients a, b can differ for different ranges of message sizes, if the memory management and communication subsystem internally switches between different protocols depending on data size. For example, CUDA transfers using DMA (cudaMemcpy) over PCIe v3, startup times between 5 and 10 microseconds for short messages are reported in the literature [4, 11]; and for some GPU types, higher transfer startups by one order of magnitude have been measured [11], which we will experimentally confirm for our system and for larger transfer sizes later in this section. Transfers to USB-connected external GPUs (e.g., via Intel Thunderbolt) may also expose longer startup times compared to PCIe3. In contrast, per-element transfer times a over PCIe v3 are typically around or even below one nanosecond nowadays. Using remote GPUs over network connections can further increase startup times. While the transfer startup time impact is expected to be rather insignificant for large operand transfers with millions of elements, they are an issue for small and medium-sized operands. Later in this section we will see that fusing two operand transfers and thus eliminating one startup time will have significant impact on overall transfer time as long as the operand sizes are well below 1M elements, and especially if transfer fusion is coupled with device memory allocation fusion, it is still effective for operand sizes around 1 million elements. While we prototype our transfer fusion optimization in this work for discrete GPUs, we expect that it may become even more important for future accelerator types with fast, multi-operand kernel computations and distributed memory.

Table 1: Time Impact of One Transfer Fusion

Vector	Transfer Fusion Only		
Length	Time/Call	Saving	Saving
[floats]	[μs]	[μs]	[%]
1K	36	6	18.9%
4K	44	8	19.7%
8K	54	4	8.5%
16K	82	2	2.6%
32K	132	15	12.0%
64K	210	17	8.2%
128K	368	13	3.8%
256K	676	70	10.4%
512K	1162	70	6.0%
1M	2313	89	3.9%
4M	9300	88	1.0%

Table 2: Average Time Impact of Allocation Fusion

Vector	Transfer Fusion Plus Fusing		
Length	Three Vector Allocations		
	Time/Call	Saving	Saving
[floats]	[μs]	[μs]	[%]
1K	404	27	6.7%
4K	418	35	8.5%
8K	423	35	6.3%
16K	448	15	3.5%
32K	496	22	4.5%
64K	570	23	4.0%
128K	1058	345	32.6%
256K	1699	737	43.4%
512K	2242	756	33.7%
1M	3409	742	21.8%
4M	10317	799	7.8%

In the following, we assume for simplicity of presentation that the same constants a and b apply for both upload and download transfers; our approach can be easily generalized for asymmetric communication costs.

Table 1 shows the saving of a single transfer fusion in a single binary-add kernel for different numbers of vector elements (floats), measured on a high-end laptop with an Intel Core(TM) i7-4710MQ running at 2.5GHz and a Nvidia K2100M (Kepler) GPU running CUDA 8 (driver version 390.87). For the same platform, Table 2 shows the times including cudaMalloc and cudaFree calls and the achieved savings due to also fusing the three operand memory allocations in addition to the transfers, for operands that are adjacent in memory. For both tables, the times are averaged over 200 repetitions, and the very time-consuming first CUDA memory allocation call is performed in a dummy allocation before the measured code fragment in order not to affect the measurement results.

We observe that, in both tables, savings in absolute times vary over the range of vector length and assume that this comes from CUDA internally switching between different memory allocation strategies and transfer protocols for small and larger vectors. The relative performance impact of transfer fusion is most significant

Table 3: Average Time Impact of One Kernel Fusion

Vector Length [floats]	No Kernel Fusion Time Per Call [μs]	Kernel Fusion Saving [μs]	Kernel Fusion Saving [%]
1K	35	5	16.4%
4K	43	6	15.7%
16K	80	8	10.0%
64K	204	9	4.6%
256K	664	22	3.7%
1M	2276	54	2.4%
4M	9236	140	1.5%

for short vectors with a few thousand elements and approaches 20% at 4K elements on our machine. For large vectors, the transfer fusion savings are one order of magnitude larger (70 to 90μs) than for short vectors (6 to 8μs), but their relative impact on overall performance is of course lower.

The saved memory allocation overhead (for both host and device memory allocations that are replaced by pointer arithmetic) is consistently positive for groups of three[1] or more consecutive vectors, where the saving is also consistently higher than the transfer fusion saving alone. The relative impact of allocation fusion is more significant at the larger vector sizes, up to 1 million elements, and decreases again for 4 million elements.

Interestingly, the literature almost only discusses *kernel* fusion, ignoring the additional optimization potential of transfer (and allocation) fusion. For comparison, we measured for a pair of binary vector-add kernels the performance impact of (parallel) kernel fusion alone (i.e., one kernel startup less), see Table 3. Comparing these figures with Table 1, we find that the kernel startup saving is, on our machine, about the same order of magnitude as the saving of one transfer fusion (not including memory allocation fusion). Taking memory allocation fusion into account, the impact of transfer fusion is higher than that of parallel kernel fusion.

3 PROBLEM FORMULATION

Machine model. We are given a heterogeneous computer system consisting of a general-purpose CPU and an accelerator (e.g., a GPU) with separate memory. Input operands of a kernel call executed on the accelerator need to be transferred from main memory to accelerator memory before kernel execution and output operands of such a kernel need to be transferred back from accelerator memory to main memory after the kernel has finished execution.

Program model. We are given a straightline-control kernel program[2] consisting of N kernel call executions (for brevity: "kernels" from now on) on a system with CPU and accelerator. We assume that the resource assignment for the kernels is known; each kernel $i = 0, ..., N-1$ is executed either on the CPU (device $d_i = 0$) or on the accelerator ($d_i = 1$).

[1]For our CUDA GPU, we also observed an anomaly for memory allocation fusion, in the case of fusing exactly *two* vector allocations for some (mostly, the smaller) vector sizes, where two separate CUDA memory allocations of size N actually perform faster than one allocation of $2N$ elements. However, this effect disappears when fusing three (as in Table 2) or more vector allocations. We can only speculate that this anomaly might be caused by some stateful optimization within the CUDA memory allocator, and it might also be specific to our GPU, CUDA and driver version.

[2]Straightline control holds for use with lazy execution [3] and for branch-free regions in a kernel-level compiler IR.

Each kernel operand is a data array stored contiguously in memory, managed by a generic data container object called *vector*. For simplicity, we assume for now that accelerator memory is always large enough to accommodate all operand data to be temporarily stored there. A generalization of our model to consider finite device memory capacity is left to future work, nevertheless we already prepare our method for such an extension by a more space-aware strategy.

The given straightline-control kernel program over vectors is assumed to be in *static single assignment form*, i.e., each vector is written at most once. Programs not fulfilling this property due to the reuse of the same vector for storing different values during program execution can be converted into single-assignment form by variable renaming.

The dependences among kernels and vectors form a dataflow graph, which we model by a directed acyclic bipartite graph, the *kernel-vector graph* $G = (K, V, E)$, where $K = \{0, ..., N-1\}$ is the (index) set of N kernels and $V = \{0, ..., M-1\}$ is the (index) set of M vectors, where $V_{init} = \{0, ..., M_{init} - 1\}$ and where $M_{init} < M$ denotes the vectors live on entry to the program, while the vectors in $V - V_{init}$ are computed by kernels in the program. An edge (v, i) exists in E iff vector v is an input operand[3] of kernel i, and an edge (i, w) exists in E iff vector w is an output operand of kernel i.

The graph is acyclic due to the single-assignment property, and each vector has at most one producer. Consequently, each vector used as an input operand of a kernel on the accelerator is also uploaded at most once and a vector used as an output operand on the accelerator is downloaded at most once.

The schedule among the kernels is given as a fixed permutation of the kernels in a topological order, i.e., the kernel call executions are totally ordered, and for simplicity of presentation we number the kernels in schedule order.

Operand transfers. The *baseline code generation strategy* is that immediately before each kernel call i to be executed on the accelerator ($d_i = 1$), we transfer (upload) each of its $indeg(i)$ input operands to the accelerator, if they are not already present. Likewise, we need to transfer (download) each of its $outdeg(i)$ output operands from the accelerator memory if it will not be further used (as input of a later kernel call) on the accelerator, and do this otherwise immediately after the last kernel call using it; thereafter we can free the output operand's space in accelerator memory.

The transfer fusion problem. Two operand transfers in the same direction can be merged if not prevented by data dependences and if the operands are adjacent in memory.[4] Our goal is to find a *memory mapping* of V, i.e., a permutation $p : V \to \{0, ..., M-1\}$ such that the M vectors are consecutively placed in memory in the order given by p and the total number of transfer startups is minimized.

[3]We abstract from the relative order of the input and output operands of a kernel as it could also be adapted by creating a clone of the kernel with a modified operand order.

[4]Transfer fusion could still have benefits if there is a small gap in between, see Li and Kessler [8]. However, we assume here for simplicity that all operands are sufficiently large so we can ignore this aspect for now.

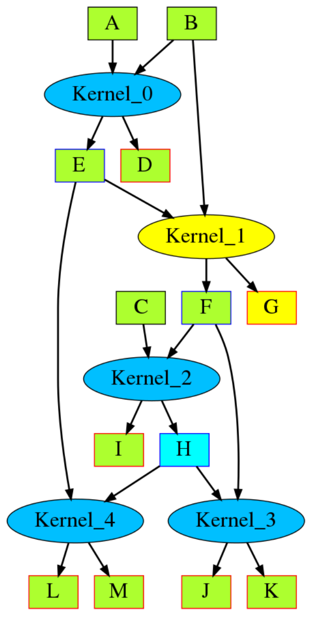


Figure 2: The kernel-vector dependence graph for a straight-line control kernel program with $N = 5$ kernels (ovals) and $M = 13$ ($M_{init} = 3$) operand vectors (boxes). Vectors 0, 1, 2... are named by uppercase letters; vectors A, B, and C are live on entry (black frame), D, G, I, L, M, J and K are live on exit (red frame). The kernel colors indicate the execution unit: yellow = CPU ($d_i = 0$), blue = accelerator ($d_i = 1$). Vectors in yellow need to be allocated on the CPU (main memory) only, vectors in green are allocated on both the CPU and accelerator, and vectors in cyanblue are allocated on the accelerator only.

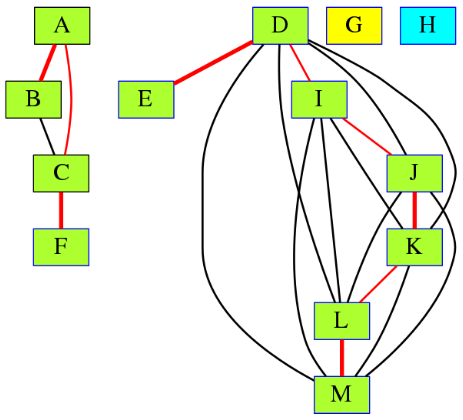


Figure 3: The affinity graph for the example of Figure 2. The edge thickness visualizes the weight (affinity). Edges shown in red belong to the best Hamiltonian paths found by our heuristic.

4 APPROACH

Given the kernel-vector dependence graph $G = (K, V, E)$ with a fixed resource mapping d and a fixed schedule, we first derive the following properties of each vector $v \in V$ used as an input or output operand of any kernel on the accelerator:

- *up_earliest*(v): The earliest relative point in time (derived from the schedule of kernels) when vector v can be uploaded. For $v \in V_{init}$, *up_earliest*$(v) = 0$, for other vectors produced on CPU but used on the accelerator it is the time step after the producer kernel.
- *up_latest*(v): The latest relative point in time when vector v can be uploaded. This is the time of the first accelerator kernel using v.
- *dn_earliest*(v): The earliest relative point in time when vector v can be downloaded. This is the time of the accelerator kernel producing v.
- *dn_latest*(v): The latest relative point in time when vector v can be downloaded. This is the time slot just before the first CPU consumer kernel of v.

Note that some vectors are both uploaded and downloaded, some are only uploaded, some are only downloaded, and some are only accessed by the CPU and need not be transferred at all.

Next, we determine the *affinity graph*, a weighted undirected graph $A = (V, E_A, a)$ with $E_A \subseteq V \times V$ and edge weights $a(\{v, w\}) \geq 0$ that indicate how much two vectors $v, w \in V$ are expected to benefit transfer fusion from being placed consecutively in memory. For that purpose, we calculate, for all vectors v, $w \in V$ to be uploaded, the overlap in time between their earliest-latest intervals

$$[up\_earliest(v){:}up\_latest(v)] \cap [up\_earliest(w){:}up\_latest(w)]$$

and, for all vectors $v, w \in V$ to be downloaded, the time overlap

$$[dn\_earliest(v){:}dn\_latest(v)] \cap [dn\_earliest(w){:}dn\_latest(w)].$$

For each such nonempty intersection encountered between any two vectors v and w, we increment the weight $a(\{v, w\})$ of affinity edge $\{v, w\}$ by 1. (For reasons explained later, we add another $\epsilon > 0$ to the weight if v and w occur as operands in the *same* kernel).

For the example kernel program in Figure 2, the following diagnostic output shows the earliest-latest intervals and their overlaps in time for the uploads (left) and downloads (right):

```
0:  A B C . . . . . . . . . .     0:  . . . D E . . . . . . . .
1:  . . C . . . . . . . . . .     1:  . . . D . . . . . . . . .
2:  . . C . . F . . . . . . .     2:  . . . D . . . . I . . . .
3:  . . . . . . . . . . . . .     3:  . . . D . . . . I J K . .
4:  . . . . . . . . . . . . .     4:  . . . D . . . . I J K L M
```

where the relative time slots (0, 1, ...) represent the kernel calls, and letters represent the vector names and possible transfer times. A dot indicates that a transfer is either not necessary or outside the earliest-latest interval. In the example, vector C could either be uploaded together with A and B at relative time 0, or instead together with F at time 2. In both cases, 2 startups could be saved, though the latter upload of C will result in lower device memory pressure between times 0 and 2, hence we give a small affinity bonus for the edge $\{C, F\}$.

Finally we remove all edges with weight 0. Figure 3 shows the resulting affinity graph for the example of Figure 2.

Next, we calculate the connected components of the affinity graph. For example, the affinity graph in Figure 3 has 4 connected components, one consisting of uploaded vectors A, B, C and F, one consisting of downloaded vectors D, E, I, J, K, L and M, and two singleton components containing vectors G (CPU-only) and H (accelerator-only), respectively.[5]

[5]Obviously, for any nontrivial scenario, at least two disjoint connected components must exist: at least one for uploaded vectors and at least one for downloaded vectors.

A (max-weight) *Hamiltonian path* in an (un)directed weighted graph is a path of $M-1$ edges $\langle v_{p(0)},...,v_{p(M-1)}\rangle$ that contains each node exactly once and maximizes the sum of weights over all edges on the path. The problem of finding Hamiltonian paths is closely related to the Traveling Salesperson Problem and is NP-complete.

For each connected component of the affinity graph with more than one vector, a maximum weight Hamiltonian path will give us a favorable linear ordering of the vectors that we can use either in forward or backward order for their memory mapping. Figure 3 shows the Hamiltonian paths for the two nontrivial connected components in red. In our prototype implementation, we use a very simple randomized heuristic[6] for computing Hamiltonian paths, based on a fixed number of affinity-biased randomized depth-first search runs generating multiple candidates among which we pick the best one to derive the memory mapping p.

Finally, we emit the resulting code. In particular, this requires going once more through the upload and download candidates (now in memory order) and deciding the relative time for each upload and download (where the earliest-latest interval is longer than 1). Where there exist multiple options, we need to break ties.

In our running example, there are two possible points in time for uploading C that lead to a minimum number of startups, either together with A and B at time 0 or together with F at time 2. Here, we either make a *greedy* choice (the earliest possible upload time maximizing the number of upload fusions at that time) or for a *tight* choice that gives preference to a late upload time which is still locally optimal within the earliest-latest interval of the considered vector but possibly leads to a globally worse solution. Choosing a later time (time 2 in our example) tends to reduce the memory pressure on accelerator device memory as life times on the device are shortened. We expose this choice as a switch to control the time-space trade-off between transfer startup saving and device memory pressure, which should be relevant in a future generalization of our approach that must respect a given device memory limit. The affinity bonus ϵ for siblings mentioned above is also intended to bias the derived mappings for the tight choice.

For our running example, the following pseudocode[7] is emitted:

```
before 0: upload B from location 0
before 0: upload A from location 1
before 0: upload C from location 2
Kernel0 (  R(B), R(A), W(D), W(E) )
after 0: download E to location 4
after 0: download D to location 5
Kernel1 (  R(E), R(B), W(F), W(G) )
before 2: upload F from location 3
Kernel2 (  R(C), R(F), W(H), W(I) )
Kernel3 (  R(F), R(H), W(J), W(K) )
Kernel4 (  R(H), R(E), W(L), W(M) )
after 4: download I to location 6
after 4: download J to location 7
after 4: download K to location 8
after 4: download L to location 9
after 4: download M to location 10
```

[6]Future work could investigate the cost-quality trade-offs of using more advanced methods for computing Hamiltonians.

[7]Our prototype implementation also emits concrete CUDA source code.

Overall, this results in a saving of 7 transfer startups (4 of which are due to the bulk download after kernel 4) compared to the baseline strategy that transfers all vector operands separately.

With the tight strategy, we obtain instead the following solution:

```
before 0: upload B from location 0
before 0: upload A from location 1
Kernel0 (  R(B), R(A), W(D), W(E) )
after 0: download E to location 4
after 0: download D to location 5
Kernel1 (  R(E), R(B), W(F), W(G) )
before 2: upload C from location 2
before 2: upload F from location 3
... (remainder as above)
```

which happens to have the same saving of 7 transfer startups. Note that the upload of C now occurs later, at time 2. Note also that the download of D is not done later by the tight choice, because E is, with the given schedule, already needed at time 1 and not fusing the transfers of D and E would thus reduce the overall saving by one startup.

5 EVALUATION

5.1 Optimization Time

We evaluate our approach with randomly generated dataflow graphs, with dataflow graphs modeling special topologies such as trees, and with dataflow graphs modeling the dependence structure of (parts of) real-world applications.

Our generator for random dataflow graphs can be configured with the number N of kernels, the number M_{init} of live-on-entry vectors, and the minimum and maximum indegree and minimum and maximum outdegree of each kernel. Kernel indegrees and outdegrees are randomly chosen within these intervals with equal probability. The overall number of vectors depends on the number of kernel outputs, i.e., is roughly linear in N. The resource assignment is also randomly chosen, with the probability for executing a kernel on accelerator ($d_i = 1$) set to 50%.

The improvement over the baseline code generation that transfers each operand in a separate message can be seen in Table 4. The first entry is the example of Figure 2. We can observe that the number of saved transfer startup times correlates with the number of vectors (which in turn is linear in the number of kernels).

Table 4 also shows the optimization times (median of 11 runs on a low-end Linux client with Intel Core i5-4250U dual-core CPU running at 2.3 GHz and having 3MB L3 cache). For the small examples with $N \leq 24$ our prototype implementation needs generally less than 1ms, and even for the largest example with 98 kernels it only needs 5.7ms. The randomized heuristic for Hamiltonians works already well in many cases, picking the best result out of only 4 attempts; increasing to 400 attempts improves the results only in a few cases (figures in parentheses) by a few more saved startups, but increases optimization time considerably. A breakdown of the optimization times for the larger test cases shows that the time for computing the Hamiltonian even with only 4 randomized DFS attempts dominates (with ca. 65%) all other steps (calculating earliest-latest, calculating the Affinity graph, and scheduling the uploads and downloads when preparing for emitting code.

Table 4: Savings in Transfer Startups and Optimization Times

N	M_{init}	Indegree min/max	Outdegree min/max	Saving, Greedy	Saving, Tight	Opt. Time
5	3	2...2	2...2	7	7	0.1ms
8	12	2...3	1...1	7 (9)	7 (8)	0.2ms
10	5	2...4	1...2	11	11	0.2ms
16	12	1...4	1...2	14	14	0.3ms
16	16	2...3	1...2	19	17	0.3ms
18	8	1...2	1...2	12	12	0.2ms
18	8	2...2	1...2	16	15	0.4ms
20	8	1...1	1...1	11	11	0.2ms
20	8	1...3	1...2	18	17	0.2ms
24	10	2...3	2...2	28	26	0.6ms
48	10	2...2	1...2	41 (42)	38 (41)	0.9ms
60	10	1...2	1...2	40	37	2.0ms
98	12	2...2	1...2	75	70	5.7ms
20	1	Linear chain		0	0	0.1ms
15	1	Out-bound bin. tree, all GPU		15	15	0.2ms
15	1	Dto., random device 50%		14	14	0.2ms
15	16	In-bound bin. tree, all GPU		15	8	0.2ms
15	16	Dto., random device 50%		14	11	0.2ms
31	32	In-bound bin. tree, all GPU		31	16	0.5ms
31	32	Dto., random device 50%		27	17	0.5ms
20	12	Horner's rule polynom. eval.		11	11	0.3ms
20	4	4x4 blocks Cholesky factoriz.		14	14	0.2ms
20	4	Dto., SYRK calls also on CPU		9	8	0.2ms
12	8	2x2 blocks Matrix multiply		11	11	0.1ms
7	2	Conj. Grad. loop, steady st.		2	2	0.1ms

We also tested some special dataflow graph topologies such as chains and trees, shown in the middle part in Table 4. For a linear chain graph with a single input vector, where every subsequent kernel depends on the preceding one, there is no transfer fusion opportunity at all, as none of the upload and download intervals overlap and hence the affinity graph contains no edges at all. For out-bound binary trees of GPU-only kernel calls starting from a single input vector only the $N+1$ downloads of live-on-exit vectors can be fused.

Finally, we consider dataflow graphs taken from real applications. For degree-N polynomial evaluation using Horner's algorithm, the resulting long narrow dataflow graph shows good optimization potential for transfer fusion that is more sensitive to the quality of the calculated Hamiltonian path than for the random graphs. The first Cholesky dataflow graph (see Figure 4) is taken from a 4×4 blocks left-looking Cholesky factorization in PLASMA [10], where we map the POTRF calls to CPU and the calls to LAPACK functions TRSM, SYRK and GEMM to GPU. In total, our algorithm saves 14 startups. Obviously, the opportunities for saving startups depend on the overall amount of necessary communication, and thus, on the mapping. For instance, if we choose, just for the sake of comparison, to run the SYRK calls on CPU instead, the saving is somewhat lower.

Likewise, we obtain relatively high startup savings for the dataflow graph of 2×2 block matrix-matrix multiply with the 8 GEMM calls mapped to GPU and 4 MADD calls to CPU, both with and without

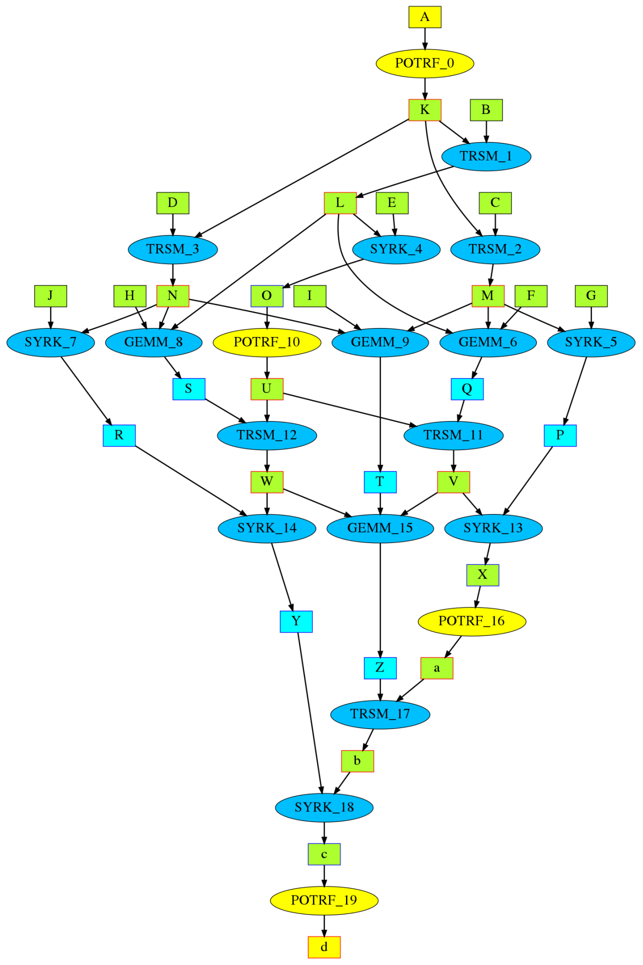


Figure 4: Dataflow graph of 4×4 blocks left-looking Cholesky factorization

manually enforcing (by increasing the corresponding affinities) that the four quarter submatrices of each input operand matrix be stored consecutively in memory.

For the same dataflow graph, different mappings lead to different optimization opportunities. For example, for the dataflow graph of the steady state of the main iterative loop in the Conjugate Gradient algorithm, with the sparse matrix-vector product call mapped to CPU and the rest to GPU, the saving is two startups.

For all considered dataflow graphs of real computations, the optimization times are low, in most cases around or below 200μs.

5.2 Overall Speedup Results

Table 5 shows, for different vector lengths, the execution times and speedups achieved by transfer fusion for CUDA source codes with synthetic kernels that were generated from the dataflow graphs in Table 4. Measurements are taken on the computer already used for the experiments in Section 2, with an Intel Core(TM) i7-4710MQ running at 2.5GHz and an Nvidia K2100M (Kepler) GPU using Ubuntu Linux 18.04 and CUDA 8. In order to reduce the impact of time variation, each code version is executed 100 times in a loop and the median time of 5 such executions is used to calculate the average time per dataflow graph code execution. Both the baseline CUDA code (No Transfer Fusion) and the CUDA code produced by our optimizer (Transfer Fusion) transfer each operand vector at most once. Allocation times are not included in these measurements, the achieved speedup is thus due to eliminated message startups only. The observed variation of execution times is very stable for small vector lengths and within a few percent for the largest vector length (4M elements). Note that our prototype implementation only works on dataflow graphs, i.e., is *not* a full compiler. The generated CUDA codes either use synthetic linear-work kernels or non-optimized CPU and GPU kernels for specific computations of non-linear work, such as GEMM. In all cases, the dependence structure and mapping of calls to GPU and CPU follows the given dataflow graph structure.

As was expected from Table 1, the measured speedups, which are due to transfer fusion alone (not allocation fusion), are largest for small vector lengths (4K and 16K floats) and decrease towards the large vector lengths. The speedup correlates with the number of saved transfer startups (see Table 4); it is largest for the random dataflow graph with 24 kernels (28 startups saved), with at best 170% speedup for vectors of 4K floats.

The results for Horner's rule polynomial evaluation, for which still 11 startups were saved, are lower because fewer vectors need be communicated in comparison, hence it is less I/O bound than the synthetic case with $N = 24$.

The benefits of transfer fusion clearly depend on the amount of work per operand element, i.e., on the arithmetic intensity of the kernels: For 4x4 Cholesky Factorization with the GEMM kernel replaced by an artificial linear-work kernel (thus simulating the case of an extremely sparse matrix), we obtain moderate speedup by transfer fusion. However, for dense matrices and GEMM calls using a cubic-work implementation, the speedup quickly drops as operand sizes increase. Likewise, Block Dense Matrix-matrix multiply is very compute-bound and does not really benefit from transfer fusion: Only if artificial linear-work kernels are used (which is the case shown in the table), there is decent potential for speedup, but if using cubic-work GEMM kernels the transfer fusion speedup drops to 0% for all sizes except the 4K element operands, for which time is reduced from 640 to 552μs (speedup 15.9%).

There is only modest speedup for the Conjugate gradient example because only two transfer startups could be eliminated.

For all examples, the transfer fusion speedups for 4M elements are within the measurement noise and can be interpreted as 0%.

Although the focus of this paper is on transfer fusion, we also have an experimental version of our prototype implementation that measures the cost and impact of memory allocation fusion guided by transfer fusion decisions. Unfortunately it can not (yet) work around the anomaly that occurs when fusing two vector allocations on our GPU system as described in Section 2, and thus negative speedups can result from allocation fusion in a number of cases, especially for small vector sizes. However, we can see good speedups for computations with low arithmetic intensity that spend, relatively, more time in CUDA memory allocation and deallocation. For illustration, we show the results of allocation (and transfer) fusion obtained for one synthetic dataflow graph, for Horner's Rule computation and for Conjugate Gradient in Table 6. We leave

Table 5: CUDA Execution Times and Speedups

Vector Length [floats]	No Transfer Fusion [μs]	Transfer Fusion [μs]	Speedup [%]
Random Dataflow Graph (N = 5, M_{init} = 3)			
4K	211	141	49.6%
16K	415	327	26.9%
64K	1136	1028	10.5%
256K	4010	3641	10.1%
1M	14455	13907	3.9%
4M	55421	55464	-0.1%
Random Dataflow Graph (N = 24, M_{init} = 10)			
4K	818	302	170.9%
16K	1850	1451	27.5%
64K	5744	5086	12.9%
256K	23188	23000	0.8%
1M	95858	94608	1.3%
4M	368109	374662	-1.7%
Horner's Rule Polynomial Evaluation (N = 20)			
4K	550	492	11.8%
16K	952	873	9.0%
64K	2567	2272	13.0%
256K	8355	7600	9.9%
1M	27836	26989	3.1%
4M	105938	105644	0.3%
4x4 Blocks Cholesky Factorization, Linear-Work Kernels			
4K	471	422	11.6%
16K	900	847	6.3%
64K	2583	2498	3.4%
256K	9119	8748	4.2%
1M	33391	33012	1.1%
4M	130660	131693	-0.8%
4x4 Blocks Dense Cholesky Factorization			
4K	543	421	29.0%
16K	1757	1586	10.8%
64K	11634	11304	0.3%
256K	85341	84500	0.0%
1M	681376	680848	0.0%
4M	6748217	6752271	-0.0%
2x2 Blocks Matrix Multiply, Linear-Work Kernels			
4K	330	256	28.9%
16K	651	572	13.8%
64K	1952	1765	10.6%
256K	7185	6505	10.5%
1M	25401	24930	1.9%
4M	100026	99363	0.7%
Conjugate Gradient loop, steady state, SPMV on CPU			
4K	148	141	5.0%
16K	271	271	0.0%
64K	769	753	2.1%
256K	2656	2581	2.9%
1M	9540	9492	0.5%
4M	37607	37411	0.5%

Table 6: CUDA Execution Times and Speedups, Including Allocations

Vector Length [floats]	No Allocation Fusion [μs]	Allocation Fusion [μs]	Speedup [%]
Random Dataflow Graph ($N = 5$, $M_{init} = 3$)			
4K	645	1165	-44.6%
16K	824	1349	-38.9%
64K	3318	2031	63.4%
256K	11373	4865	133.8%
1M	29229	14889	96.3%
4M	98861	95382	3.6%
Horner's Rule Polynomial Evaluation ($N = 20$)			
4K	891	1106	-19.4%
16K	2020	1787	13.0%
64K	5756	4738	21.5%
256K	18437	13951	32.2%
1M	28265	23890	18.3%
4M	67899	65024	4.4%
Conjugate Gradient loop, steady state, SPMV on CPU			
4K	557	850	-34.5%
16K	649	999	-35.0%
64K	1871	1829	2.3%
256K	6523	5583	16.8%
1M	17224	16455	4.7%
4M	60741	59841	1.5%

the development of further technical improvements and a more systematic evaluation of allocation fusion to future work.

6 RELATED WORK

6.1 Transfer Fusion

While kernel fusion has been investigated by several papers in the past decade, there appears to be almost no published work on transfer fusion and memory allocation fusion for heterogeneous systems. Li et al. [8] describe a runtime optimization that performs lazy memory placement of temporary vectors for a single kernel call to optimize transfer fusion. The general advantage of runtime methods is that they also work for statically unknown source operands (e.g. in the case of aliasing).

The global optimization method presented in this paper could, in principle, be likewise applied as a runtime optimization once sufficiently large kernel (sub)graphs such as lineages [3] have been identified at runtime, which in turn is done by lazy execution techniques that are also applied, e.g., in Spark and TensorFlow. However, in our case the runtime overhead for the optimization might only pay off if the computed memory placement can be reused, e.g. in multiple iterations over the kernel program.

6.2 Connection to Kernel Fusion

Kernel fusion is a program transformation for heterogeneous computations that merges multiple kernels and kernel calls into a single one. The purpose of this coarsening of the granularity of accelerator usage is to either improve data locality, or reduce kernel startup overhead, or improve the overall throughput by combining memory-bound with arithmetic-bound kernels. Kernel fusion is a special case of the classical *loop fusion* transformation, namely, for the case of parallel loops executing on an accelerator with many parallel hardware threads, such as a GPU.

There are two main types of kernel fusion, see also Figure 5 for illustration. With *serial fusion*, the fused kernel consists in the same set of accelerator threads executing the computations of the (possibly, dependent) original kernels in series. *Parallel fusion* refers to the co-scheduling of the computations of (originally, independent) kernels within a common kernel, which is executed by the union of the sets of accelerator threads used for each subkernel; these branch internally to the different subkernels again.

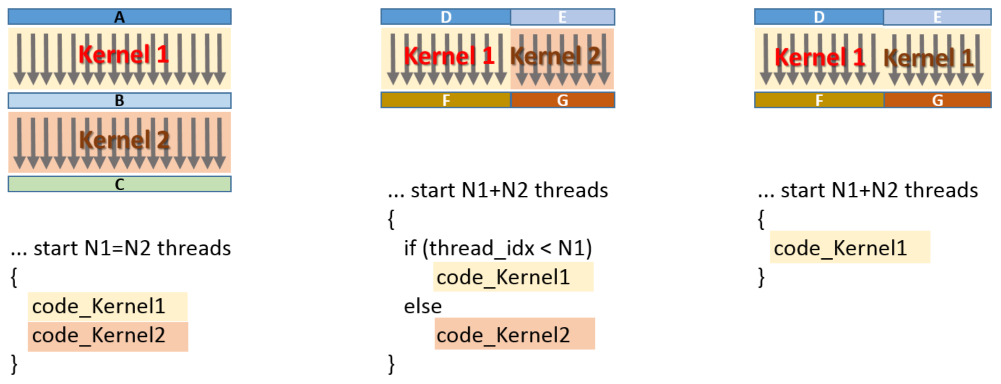
Serial fusion of dependent kernels in a producer-consumer dependence chain can be particularly effective where it allows to internalize the inter-kernel flow of bulk operand data (i.e., intermediate (sub-)vectors or -matrices) between producer and consumer kernels. This reduces the lengths of the live ranges between production and consumption for such data elements and allows to allocate these data elements in registers or in fast on-chip memory instead. Consequently, it reduces the volume of off-chip memory accesses and increases the arithmetic intensity of the resulting computation.

In contrast, *parallel fusion* does not change the arithmetic intensity [16] of the code, but eliminates kernel startup time overhead (see also our measurements in Table 3). It can also improve the thread occupancy and thus utilization of the accelerator especially for kernels with relatively small operands. Moreover, it can lead to overall improved throughput by co-scheduling memory-bound with arithmetic-bound kernels [15]. For GPUs, parallel fusion can be done at the granularity of individual threads or of thread blocks, the latter of which should give better performance [15].

The special case of parallel fusion for the *same* kernel (see the right-hand side scenario in Figure 5) constitutes the combination of parallel kernel fusion with transfer fusion, as it likewise requires that all element-wise accessed operands are adjacent in memory. This optimization is applicable e.g. in deep learning computations that can benefit from batching multiple BLAS kernel calls on multiple small operands. For instance, CUBLAS provides batched versions of BLAS kernels that can be (manually) used in such cases.

A number of automatic static kernel fusion techniques, especially for GPUs, have been presented in the literature [6, 13, 14]. Filipovic and Benkner [5] evaluate the effectiveness of parallel kernel fusion on GPU, Xeon Phi and CPU for linear algebra computations. The just-in-time compilation approach described in Wen *et al.* [15] tries to pair memory-bound with arithmetic-bound kernels in parallel kernel fusion, in order to speed up execution beyond merely saving kernel startups. Qiao *et al.* [12] consider serial kernel fusion for image processing DSLs.

None of this work explicitly studies the opportunities and effect of *transfer* fusion on parallel kernel fusion. In the present paper we have described a transfer and allocation fusion optimization that is, basically, independent of parallel kernel fusion. It could, for instance, be applied after a preceding parallel kernel fusion pass, hence the kernels in the dataflow graph exposed to our transfer and allocation fusion optimizer will already be fused where deemed profitable. Transfer fusion is still applicable in cases where parallel kernel fusion is not; for instance, some transfers of operands of subsequent kernels may be fused also in the presence of a data dependence



```
... start N1=N2 threads
{
    code_Kernel1
    code_Kernel2
}
```

```
... start N1+N2 threads
{
   if (thread_idx < N1)
       code_Kernel1
   else
       code_Kernel2
}
```

```
... start N1+N2 threads
{
   code_Kernel1
}
```

Figure 5: Left: Serial kernel fusion by sequencing code from (calls to) different kernels in the same parallel loop, preserving per-element data flow dependences between kernels in the fused code. — Middle: Parallel kernel fusion by co-scheduling two previously independent calls to different kernels within the same "superkernel". — Right: A special case of parallel fusion where both kernel calls are instances of the same kernel. Adapted from Wang *et al.* [14].

between subsequent kernels, which prevents their parallel kernel fusion.

6.3 Using Hamiltonians for Optimized Memory Mapping

Maximum-weight Hamiltonian paths in affinity graphs have also been used for solving the so-called *offset assignment problem* in generating optimized target code for DSP processors with address generation units, which allow the execution of autoincrement / autodecrement operations on one or several address register(s) in parallel with memory accesses. For a given (static) memory access sequence by the program code, a placement of scalar variables on the stack is determined that maximizes the number of subsequent accesses to variables with neighboring addresses, and hence minimizes the use of additional processor cycles for calculating non-neighbored addresses. This problem was first described by Bartley [1], together with a greedy heuristic algorithm for computing maximum weight Hamiltonian paths that was later improved by Liao et al. [9] and Leupers and Marwedel [7]. (Note that we do not claim that the simple DFS-based Hamiltonian heuristic used in our proof-of-concept prototype, albeit fast, performs on-par with these heuristics, and might well complement it by one of these heuristics in the future to better support optimization quality vs. speed trade-offs.) The compiler research community has produced much subsequent work on the offset assignment problem and its generalizations, see e.g. Liao et al. [9] or Leupers and Marwedel [7], to name only a few. The integrated solution of the combined problems of offset assignment and instruction scheduling has also been investigated, see e.g. Eriksson and Kessler [2].

7 CONCLUSION AND FUTURE WORK

We show that transfer fusion and memory allocation fusion for small and medium-sized vector operands in multi-kernel programs can have a significant speedup effect on performance as long as the program's execution time is not dominated by kernel execution times, i.e., for calculations with low operational intensity [16].

We have presented a global operand transfer fusion optimization technique for straight-line single-assignment kernel programs in heterogeneous systems. Our approach analyzes the kernel-vector dependence graph for joint upload/download affinities between vectors and calculates Hamiltonian paths to derive a reordering of the vectors in memory in order to maximize the number of data transfer fusions, and thus, minimize the accumulated transfer startup costs. Our implementation uses polynomial-time heuristics for graph analysis and Hamiltonian computation that result in low optimization times even for kernel programs with about 100 kernels and vectors. For randomly generated synthetic graphs we have shown that significant amounts of transfer fusions can be achieved, which, by and large, grow with the number of vectors (which in turn correlates with the number of kernels).

We also identified a time-space trade-off considering the lengths of live ranges of operands stored temporarily in accelerator device memory. By an affinity bonus for joint late uploads (and also for joint early downloads, which is not implemented yet) and breaking ties accordingly in emitting the final bulk-transfer code, we allow to trade shorter live ranges in device memory for a possibly slightly lower amount of saved transfer startups.

The presented optimization could be integrated in a compiler for kernel-based programs. It could also be used as a just-in-time optimization, e.g., in a runtime system for kernel programs, such as StarPU or OmpSS, especially where the considered kernel program is part of an iterative loop not affecting the dependence structure, such that the one-time optimization cost can be amortized over many iterations while the gains in terms of reduced message startups will pay off in every iteration. For a runtime optimization scenario, the vector size up to which the optimization of transfer startups remains significant can be determined once for the target system by microbenchmarking, so that the optimization is automatically skipped for computations on larger vectors.

Future work could extend the experimental evaluation by including further GPU types (and possibly other accelerators), by also fully evaluating the effect of memory allocation fusion beyond transfer fusion, and by including more kernel dataflow graphs from real-world programs e.g. from deep learning. The current prototype implementation can certainly be improved; beyond the ongoing extension of the CUDA source code generator, the simplistic DFS-based Hamiltonian heuristic could be replaced by more elaborated Hamiltonian heuristics from the literature.

Future work can extend and generalize the problem setting and its solution in various ways. The combination of transfer fusion with kernel fusion seems natural and needs be studied in more detail; in particular, possible interferences and trade-offs need to be investigated. The combination of the vector reordering in memory with a *rescheduling* of the kernels (as far as permitted by the data dependences in G) will increase flexibility, which can permit additional optimization opportunities but is also expected to increase the optimization time. Also the resource mapping could be co-optimized with transfer fusion because there is a cyclic interdependence, too—the transfer problem depends on the resource mapping, but the cost model for the mapping problem should, where significant, also take the fusion effects into account. Another possible extension is a space-aware optimization performing device memory assignment and keeping a given device memory limit. Also, a pre-placement of some of the vectors could be given, e.g., for live-on-entry vectors. Finally, for the use as a static optimization in compilers, we could work towards relaxing the straightline-control constraint.

ACKNOWLEDGMENTS

This work has been partly supported by EU H2020 project EXA2PRO (801015).

The author thanks August Ernstsson and Lu Li for discussions about technical aspects of this work.

We also thank the anonymous reviewers for their constructive feedback, which helped to improve this paper.

REFERENCES

[1] David H. Bartley. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exp.* 22, 2 (1992), 101–110.

[2] Mattias Eriksson and Christoph Kessler. 2011. Integrated Offset Assignment. In *Proc. 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, George Cai and Tom van der Aa (Eds.). Chamonix, France, 47–54.

[3] August Ernstsson and Christoph Kessler. 2018. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience* (2018). (to appear).

[4] Yusuke Fujii et al. 2013. Data transfer matters for GPU computing. In *Proc. ICPADS'13*.

[5] Jiri Filipovic and Siegfried Benkner. 2015. OpenCL kernel fusion for GPU, Xeon Phi and CPU. In *Proc. 27th Int. Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD'15)*. IEEE, 98–105. https://doi.org/10.1109/SAC-PAD.2015.29

[6] Jiri Filipovic, Matus Madzin, Jan Fousek, and Ludek Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71 (2015), 3934–3957. https://doi.org/10.1007/s11227-015-1483-z

[7] Rainer Leupers and Peter Marwedel. 1996. Algorithms for address assignment in DSP code generation. In *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD'96)*. IEEE Computer Society, Washington, DC, USA, 109–112.

[8] Lu Li and Christoph Kessler. 2018. Lazy Allocation and Transfer Fusion Optimization for GPU-based Heterogeneous Systems. In *Proc. PDP'18 conference*. IEEE, Cambridge, UK, 311–315. https://doi.org/10.1109/PDP2018.2018.00054

[9] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. 1996. Storage assignment to decrease code size. *ACM Trans. on Progr. Lang. and Syst.* 18, 3 (1996), 235–253.

[10] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2010. A scalable high performant Cholesky factorization for multicore with GPU accelerators. In *Proc. 9th international conference on High performance computing for computational science (VECPAR'10)*. Springer, 93–101.

[11] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization. In *Proc. HPCA'13*.

[12] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2018. Automatic kernel fusion for image processing DSLs. In *Proc. 21th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES'18)*. ACM. https://doi.org/10.1145/3207719.3207723

[13] Mohaned Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *Proc. Int. Conf. for High-Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 191–202. https://doi.org/10.1109/SC.2014.21

[14] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In *Proc. IEEE/ACM Int. Conf. on Green Computing and Communications and Int. Conf. on Cyber, Physical and Social Computing*. 344–350. https://doi.org/10.1109/GreenCom-CPSCom.2010.102

[15] Yuan Wen, Michael F.P. O'Boyle, and Christian Fensch. 2018. MaxPair: Enhance OpenCL concurrent kernel execution by weighted maximum matching. In *Proc. GPGPU-11*. ACM. https://doi.org/10.1145/3180270.3180272

[16] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76.