Global Optimization of Operand Transfer Fusion in Heterogeneous Computing

Christoph Kessler

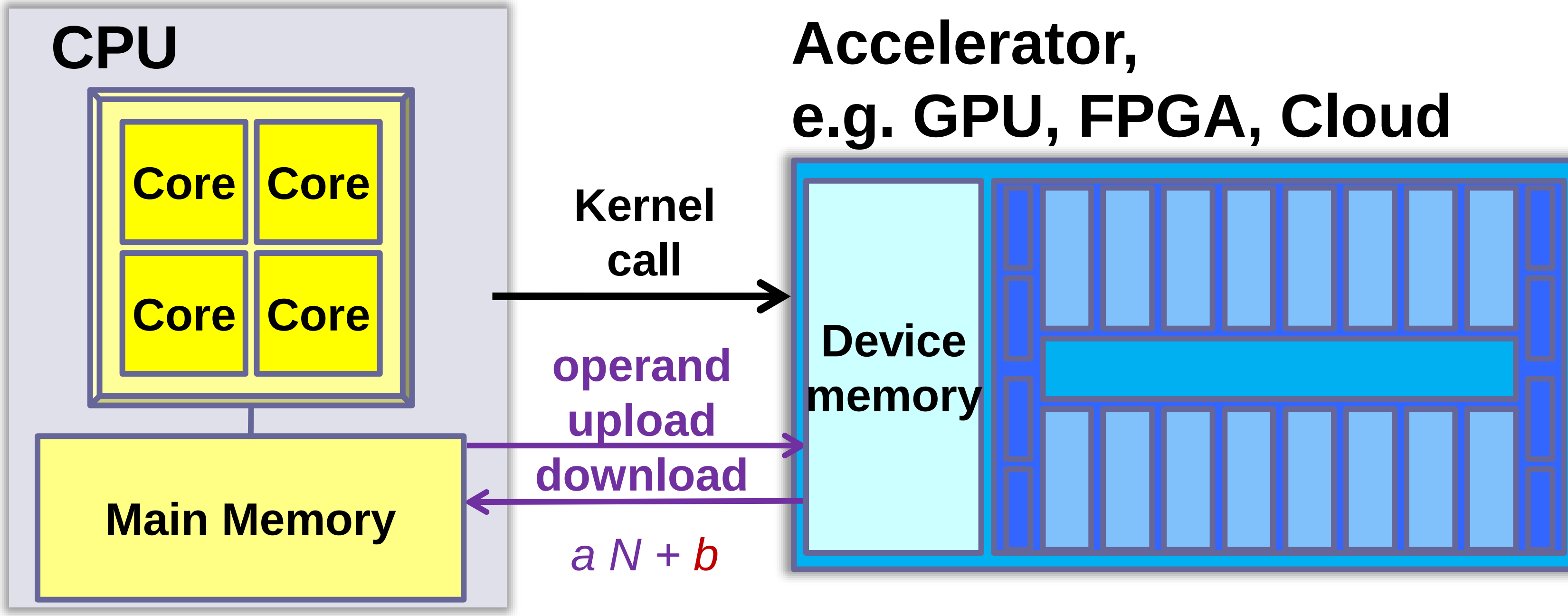
Linköping University
Sweden

LINKÖPING UNIVERSITY

Presented at
SCOPES-2019
St. Goar, Germany

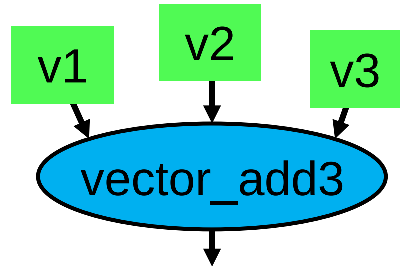
exa2pro

Heterogeneous Systems with Distributed Memory



- Distributed memory, explicit operand transfers
- High data transfer cost (esp. over PCIe / IP...): $a\ N + b$
  - High startup cost
    significant for small messages, e.g. $b\ /\ a \sim 10^4$
- Goal: Message fusion for operand transfers
  → reduce #startups

Example: 1 kernel call, 3 input operands



(a) **Arbitrary operand order in memories:**

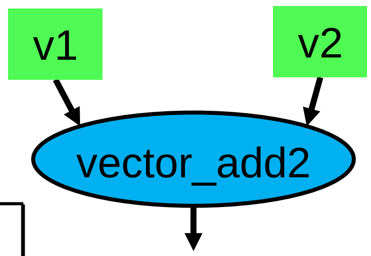
```
float *v1 = malloc(N); ...
float *v2 = malloc(N); ...      3 calls
float *v3 = malloc(N);

cudaMalloc( &g_v1, N ); ...
cudaMalloc( &g_v2, N ); ...     3 calls
cudaMalloc( &g_v3, N );

...                3 upload messages,
                   time 3(aN+b)

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

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

(b) **Operands consecutive in both memories:**

```
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;               Allocation fusion

...                1 upload message,
                   time 3aN + b

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

                                Transfer fusion

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

Saving Potential on a Concrete GPU
By Transfer Fusion

Nvidia Kepler K2100, CUDA 8, driver 390.87



Binary Vector-Add
(1 Transfer Fusion)

Vector Length [floats]	Transfer Fusion Only		
	Time/Call [μs]	Saving [μs]	Saving [%]
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%

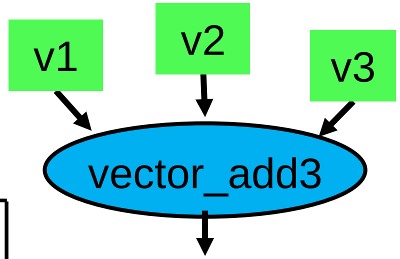
- Memory allocation time not included
- Decent relative savings for up to 1M float elements (at low arithmetic intensity)

Saving Potential on a Concrete GPU
By Transfer and Allocation Fusion

Nvidia Kepler K2100, CUDA 8, driver 390.87

Ternary Vector-Add

(**2** Transfer + **Allocation** Fusions)

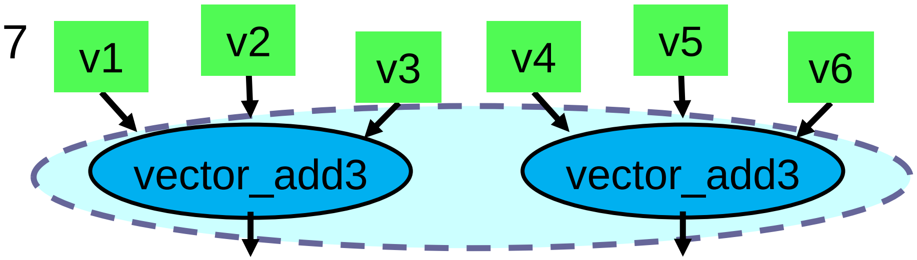


Vector Length [floats]	Transfer Fusion Plus Fusing Three Vector Allocations		
	Time/Call [μs]	Saving [μs]	Saving [%]
1K	404	27	6.7%
4K	418	35	8.5%
8K	423	35	6.3%
16K	448	15	3.5%
32K	496	22	4.5%
64K	570	23	4.0%
128K	1058	345	32.6%
256K	1699	737	43.4%
512K	2242	756	33.7%
1M	3409	742	21.8%
4M	10317	799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

- Memory allocation time included (except a dummy first cudaMalloc)
- Decent relative savings for up to 4M elements (at low arithmetic intensity)
- Largest impact observed for medium-length operands
- Anomaly observed for *binary*-add: slowdown for small operand sizes 1 x cudaMalloc(2N) can be slower than 2 x cudaMalloc(N)
  - cause is unclear (stateful specul. optimization in cudaMalloc?)

Saving Potential on a Concrete GPU
Parallel Kernel Fusion on-Par with Transfer Fusion

Nvidia Kepler K2100, CUDA 8, driver 390.87

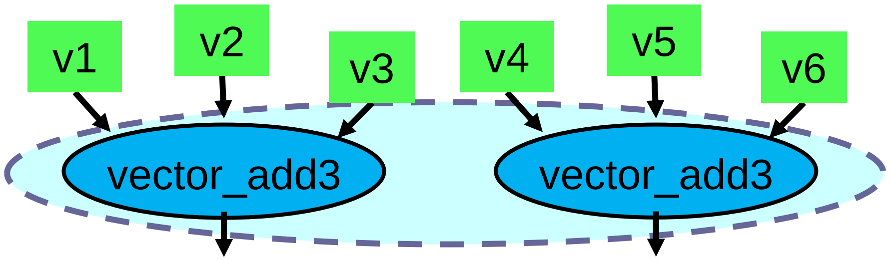


Ternary Vector-Add
(2 Transfer Fusions
+ Parallel Kernel Fusion)

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%

- Memory allocation time not included
- Operands (v1+v4, ...) consecutive in memory in both cases
- Decent relative savings for up to 1M float elements (at low arithmetic intensity)
- Speedups in the same order of magnitude as by transfer fusion

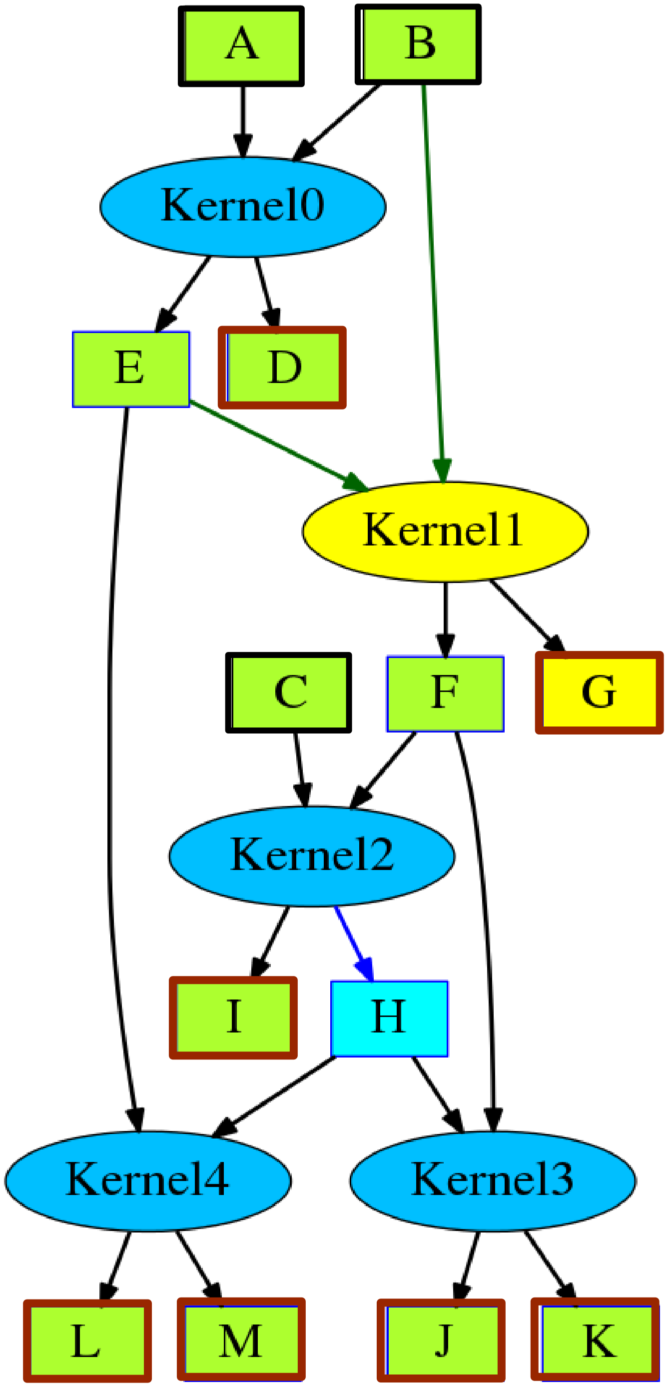
Motivation



- Transfer fusion gain (kernel startup time) on our system almost as high as parallel kernel fusion gain (kernel launch time)
- Parallel kernel fusion is not always applicable/beneficial, but transfer fusion may still apply
- We will focus on **transfer fusion** in this work.
  - Objective: Maximize # transfer fusions in a program
    - Global scope → more fusion options across multiple kernel calls
  - Allocation fusion as side effect may give further speedup (where not suffering from the single-fusion anomaly of our GPU)

Global Scope of Allocation

- **Kernel-Vector Data Flow Graph**
  - *Call nodes*: Kernel0, Kernel1, ...
    - Synchronous calls
    - *Mapping* to device/host given
    - Fixed *schedule* given
      → trace of calls, relative time: 0, 1, ...
  - *Vector (data) nodes*: A, B, C, ...
    - Static single assignment
    - Some live-on-entry, some live-on-exit
  - Data flow edges
  - No control flow
- Base-line code generation:
  Each vector transfered to/from device at most once, in a separate message



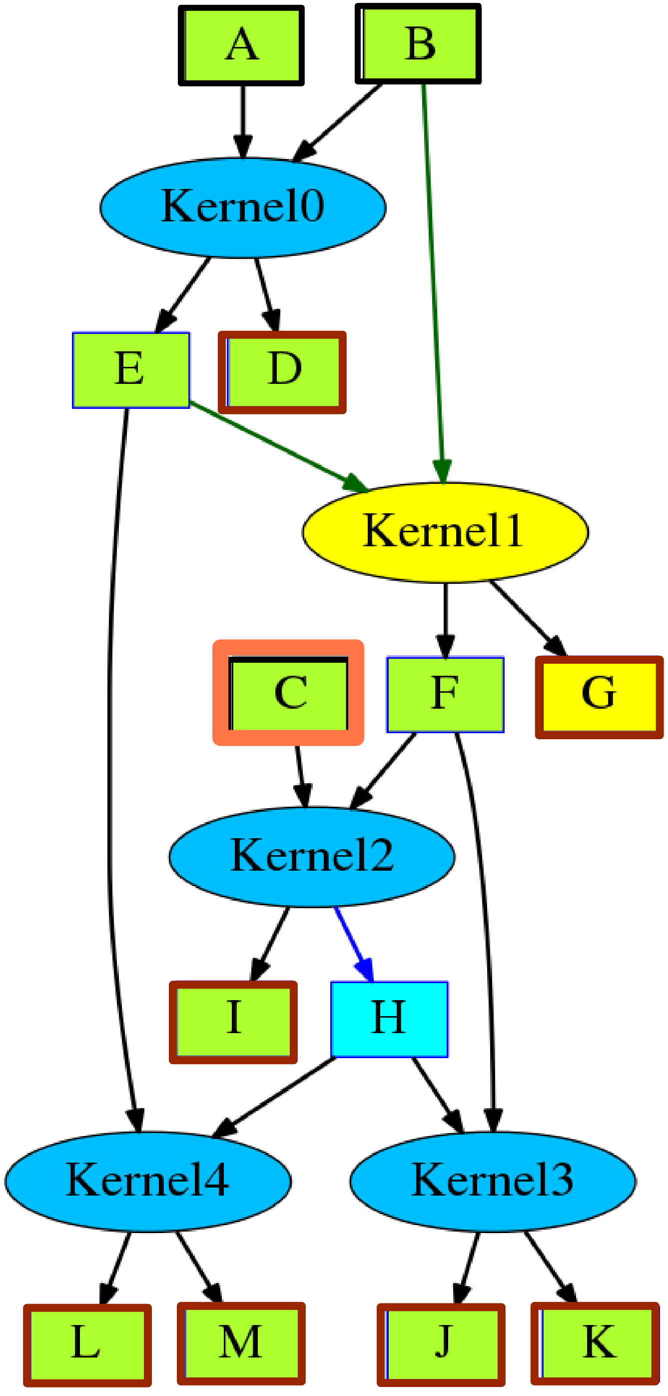
Global Scope of Allocation

- Kernel-Vector Data Flow Graph (fixed schedule, fixed mapping)
- **Calculating earliest and latest time points for uploads and downloads:**
  - depend e.g. on relative time of the producing resp. earliest consuming kernel calls
    - Details in the paper

Uploads (earliest ... latest):

← Flexibility

```
0:  A B C . . . . . . . . .
1:  . . C . . . . . . . . .
2:  . . C . . F . . . . . .
3:  . . . . . . . . . . . .
4:  . . . . . . . . . . . .
```



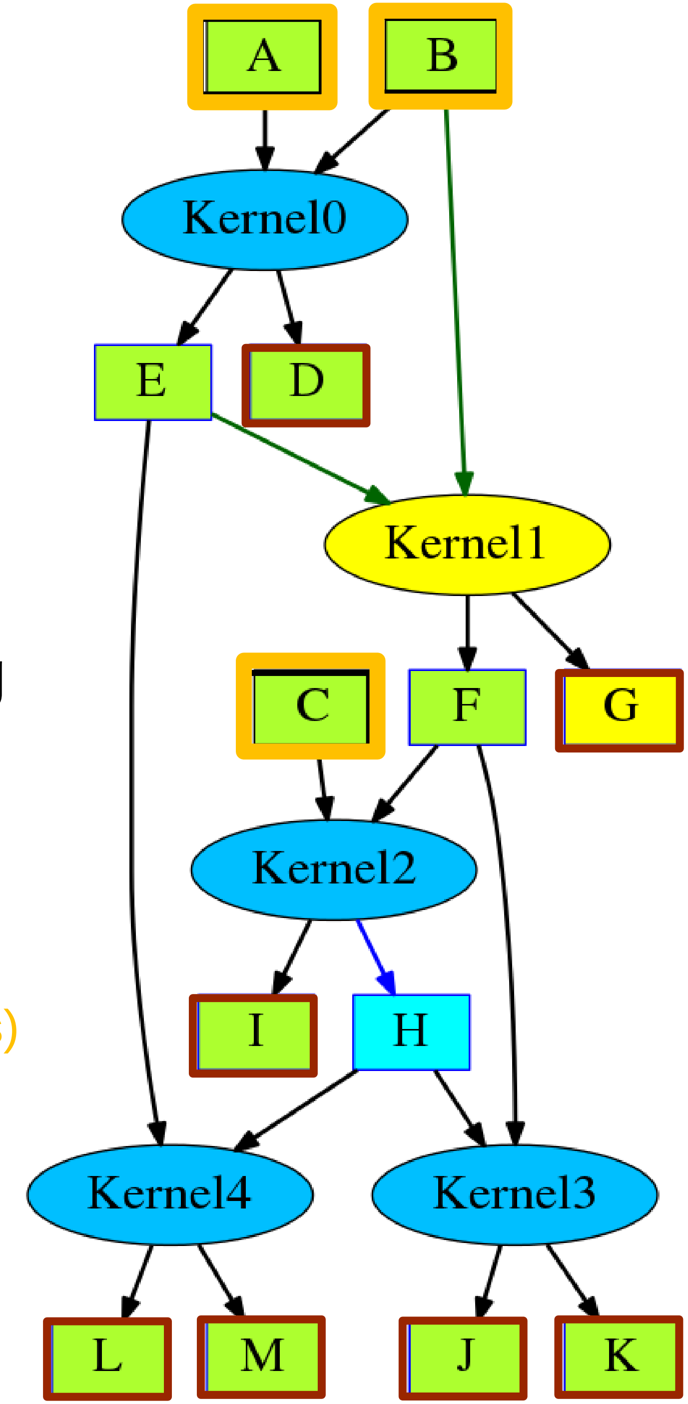
Global Scope of Allocation

- Kernel-Vector Data Flow Graph (fixed schedule, fixed mapping)
- **Calculating earliest and latest time points for uploads and downloads:**
  - depend e.g. on relative time of the producing resp. earliest consuming kernel calls
    - Details in the paper

Uploads (earliest ... latest):

```
0:  A B C . . . . . . . . .
1:  . . C . . . . . . . . .
2:  . . C . . F . . . . . .
3:  . . . . . . . . . . . .
4:  . . . . . . . . . . . .
```

← Fusion possibility (2 startups)

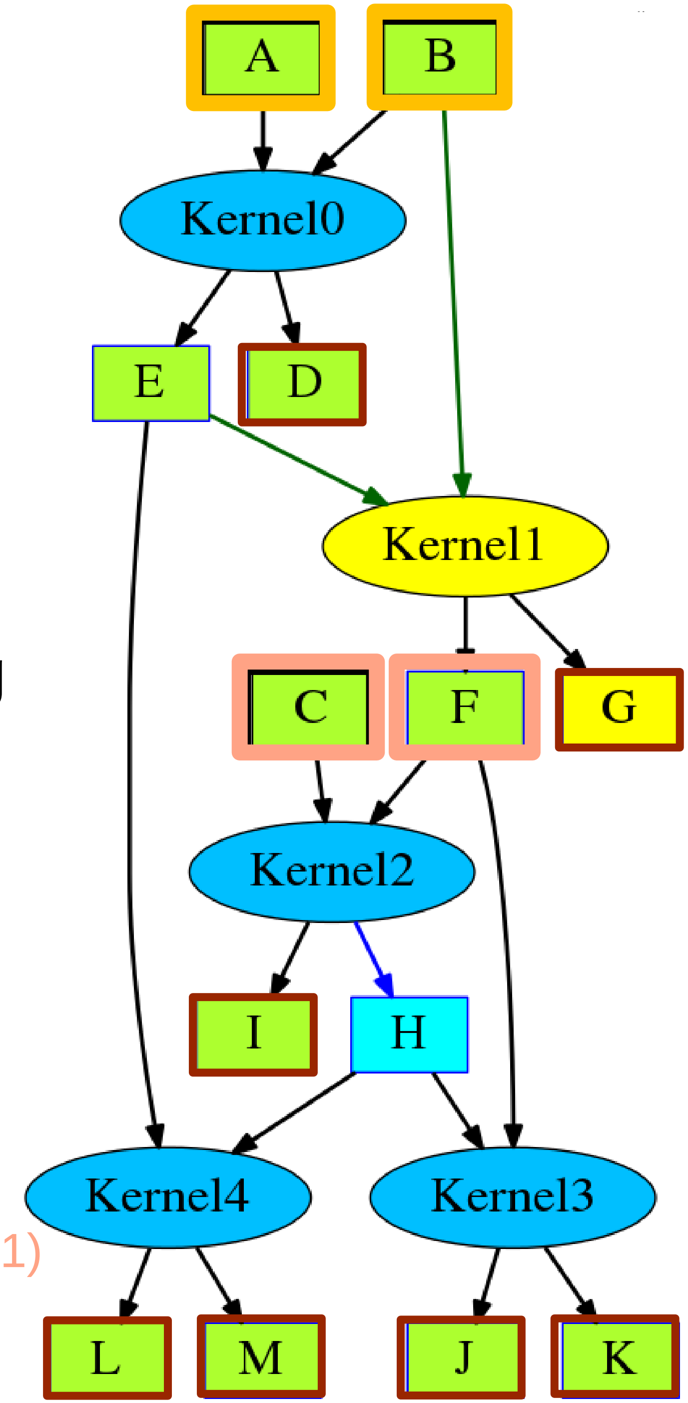


Global Scope of Allocation

- Kernel-Vector Data Flow Graph (fixed schedule, fixed mapping)
- **Calculating earliest and latest time points for uploads and downloads:**
  - depend e.g. on relative time of the producing resp. earliest consuming kernel calls
    - Details in the paper

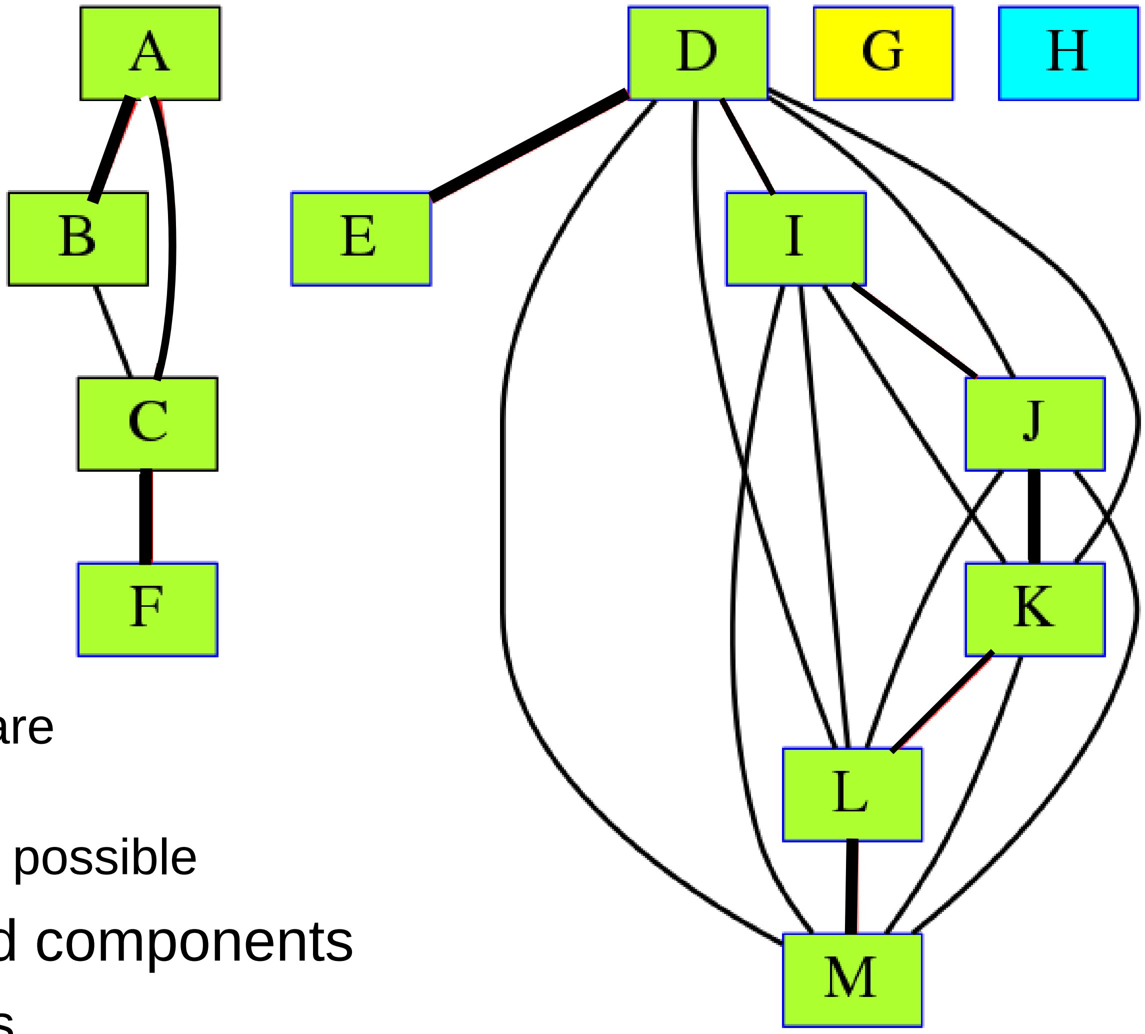
Uploads (earliest ... latest):

```
0:  A B C . . . . . . . . .   ← Fusion possibility (1)
1:  . . C . . . . . . . . .
2:  . . C . . F . . . . . .   ← Fusion possibility (1)
3:  . . . . . . . . . . . .
4:  . . . . . . . . . . . .
```



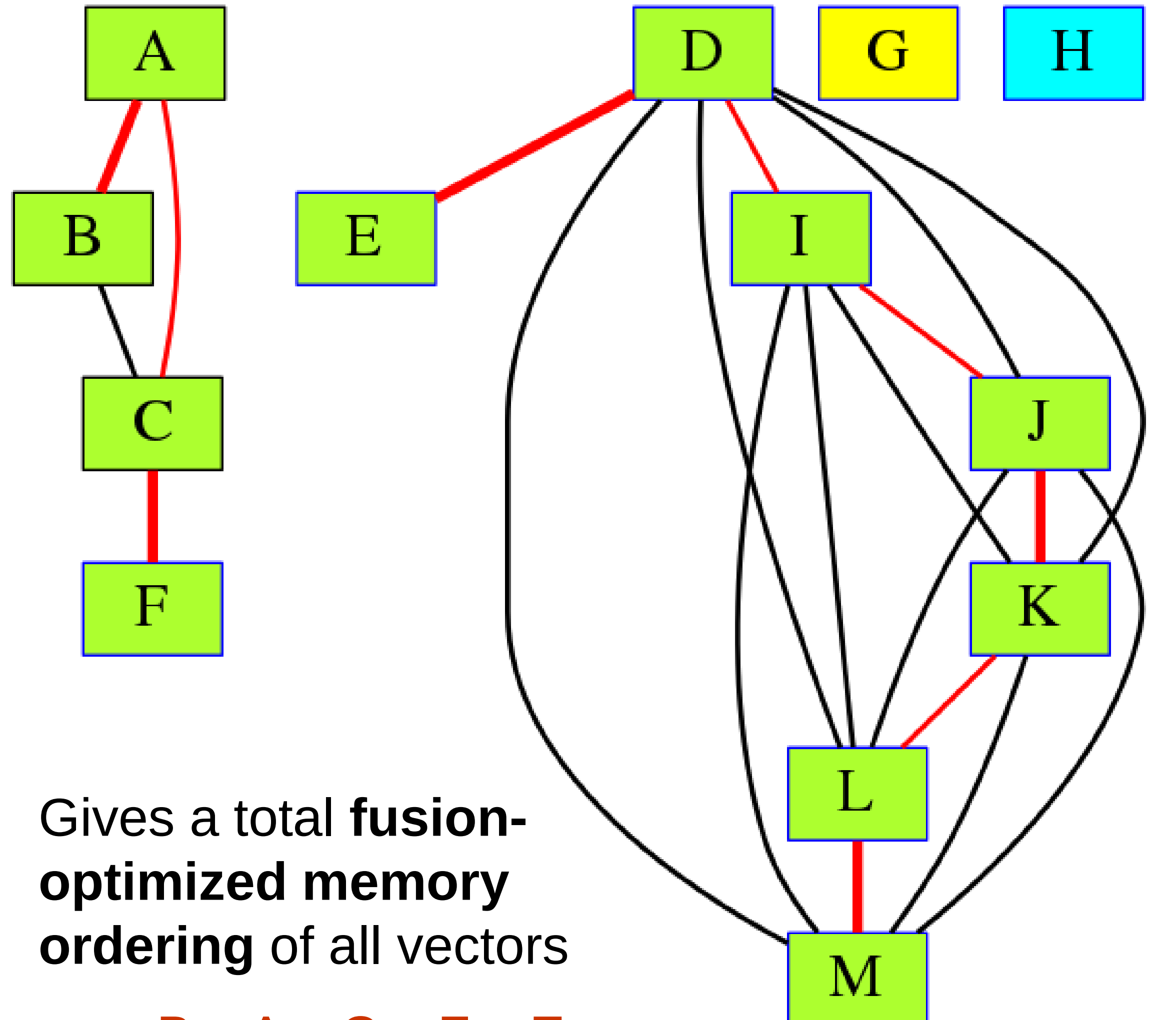
Affinity graph

- Nodes = vectors
- Undirected edges
  { *u*, *v* } with weight
  = **affinity** of *u*, *v*
  - expected fusion gain of allocating *u*, *v* consecutive in memory
- By finding overlapping earliest-latest intervals
  - +1.0 if uploads resp. downloads of *u* and *v* can be fused if *u* and *v* are consecutive in memory
  - Bonus for tight solutions possible
- Has at least 2 connected components
  - $\geq$1 for uploaded vectors,
    $\geq$1 for downloaded vectors



Affinity graph, Max-Weight Hamiltonian Paths

- **Hamiltonian path** of a graph = a path of *n*-1 edges visiting each of the *n* nodes exactly once
- Goal: Find a **maximum-weight** Hamiltonian path in each connected component of the affinity graph
- NP-complete (related to TSP)
- Linear-time heuristic based on DFS (see paper)

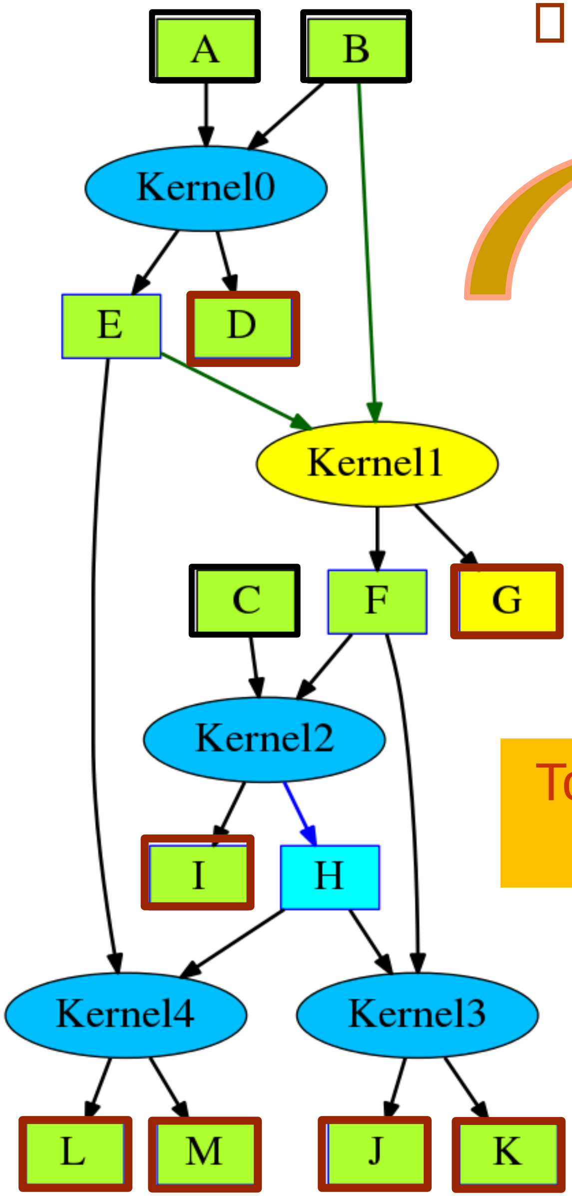


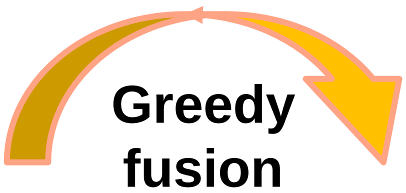
Gives a total **fusion-optimized memory ordering** of all vectors

e.g. B – A – C – F – E – D – I – J – K – L – M

Emitting the Fused Code

- optimized ordering:
  B – A – C – F – E – D – I – J – K – L – M

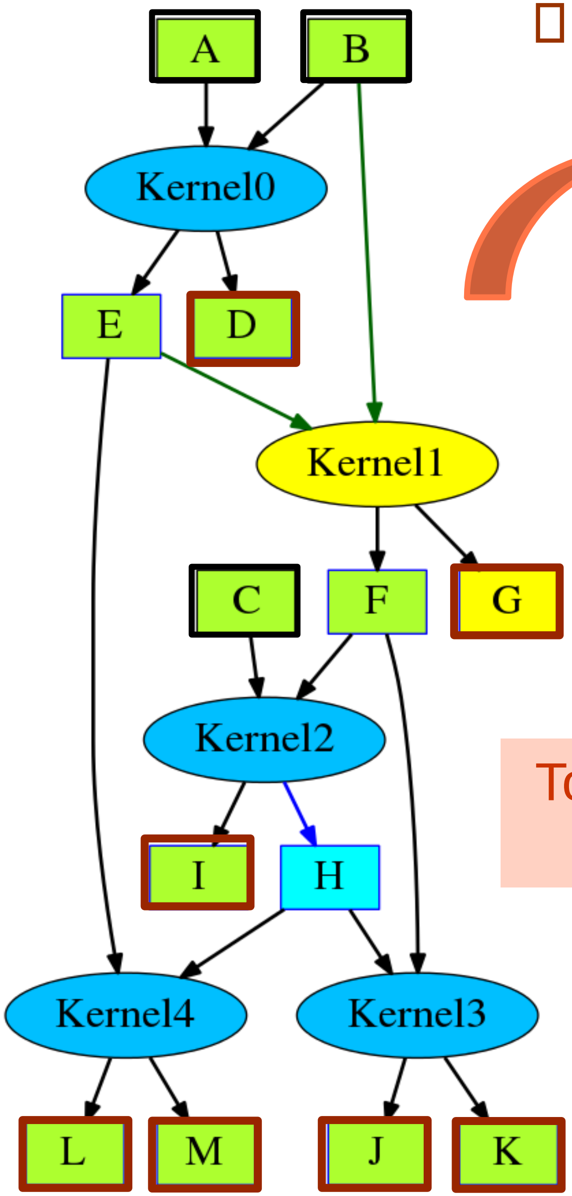





```
before 0: upload B from location 0
before 0: upload A from location 1
before 0: upload C from location 2
Kernel0 (  R(B), R(A), W(D), W(E) )
after 0: download E to location 4
after 0: download D to location 5
Kernel1 (  R(E), R(B), W(F), W(G) )
before 2: upload F from location 3
Kernel2 (  R(C), 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
```

Total savings:
7 startups

Emitting the Fused Code, Alternative



- optimized ordering:

B – A – C – F – E – D – I – J – K – L – M

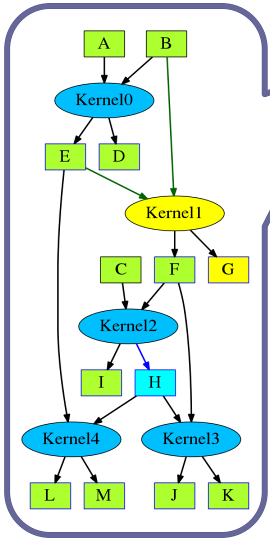
Tight fusion

```
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
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
```

Total savings:
7 startups

Later uploads → lower memory pressure on device

Evaluation of # Saved Transfer Startups and Optimization Time



Savings in comparison to single operand vector uploads / downloads (no fusion)

Table 1: Synthetic Programs, Savings in Transfer Startups

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

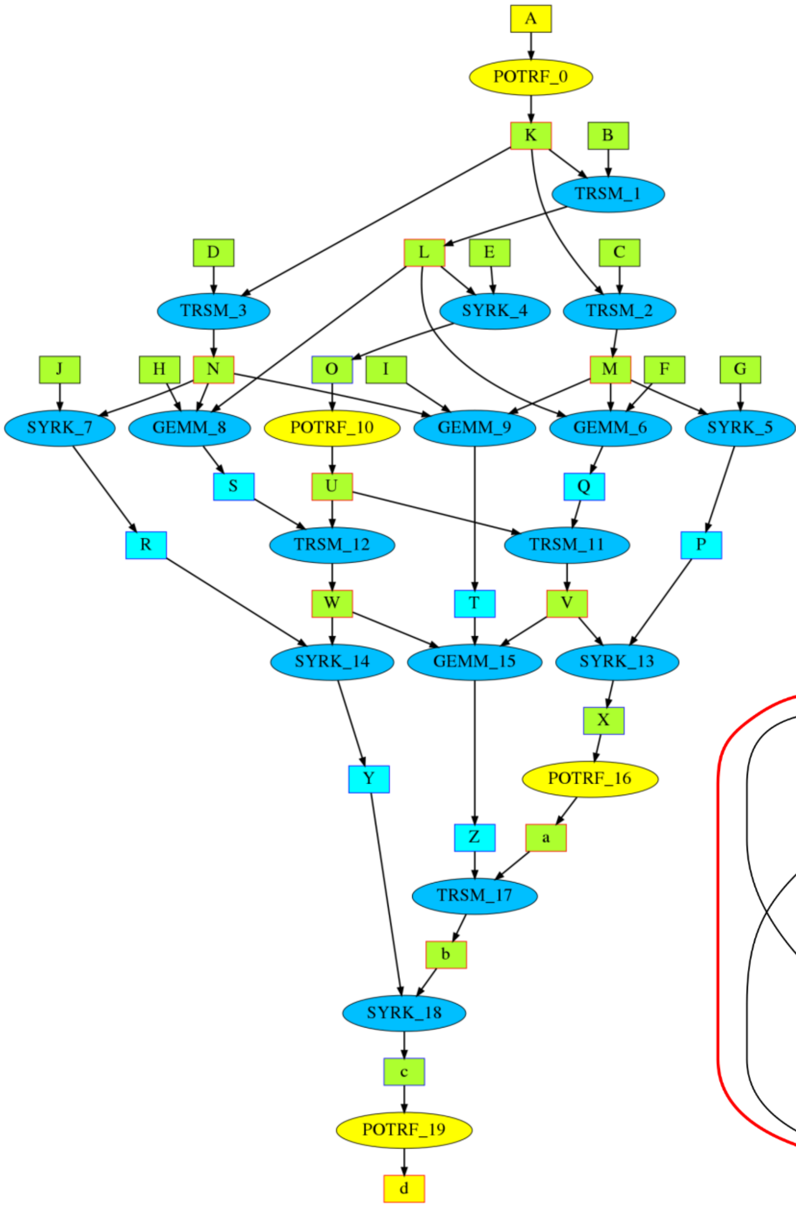
(in parentheses: savings with higher-effort Hamiltonian heuristic)

Evaluation of # Saved Transfer Startups and Optimization Time, cont.

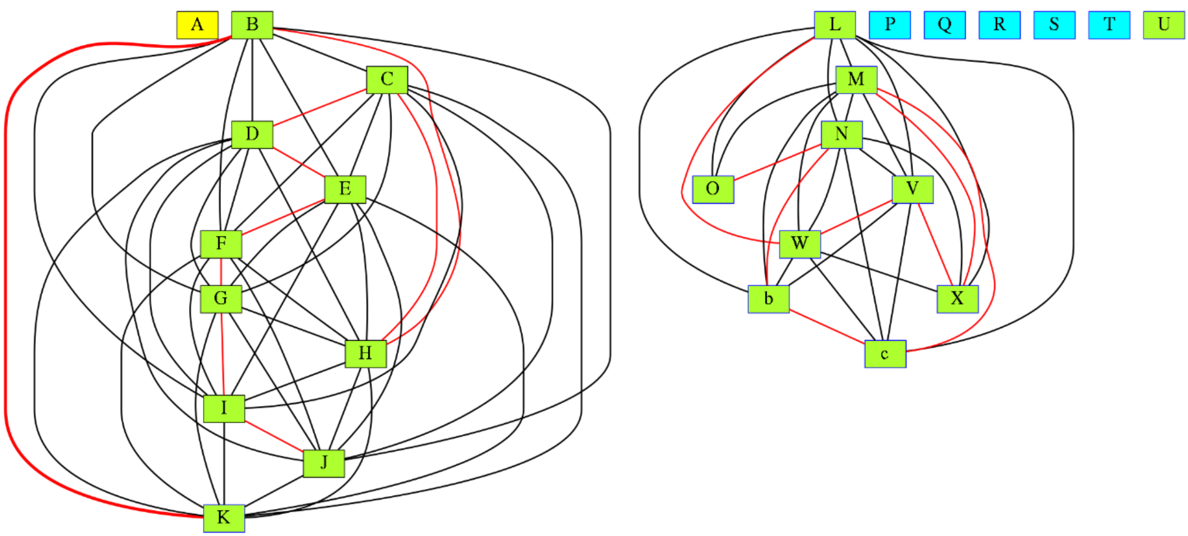
Special task graph topologies and application-derived task graphs

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

4 x 4 Block Cholesky Factorization



Affinity Graph:

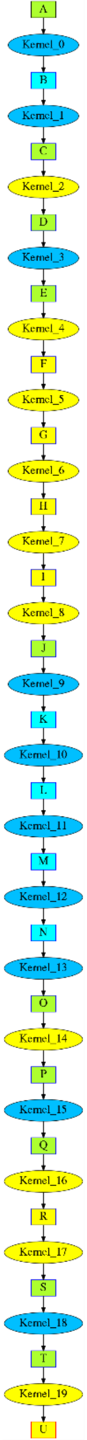


Evaluation of # Saved Transfer Startups and Optimization Time, cont.

LINKÖPING UNIVERSITY

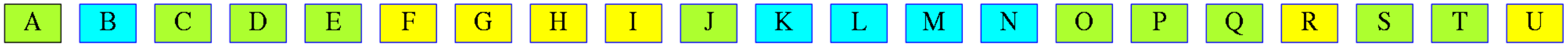
Special task graph topologies and application-derived task graphs

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



Linear Chain (N=20)

Affinity Graph:

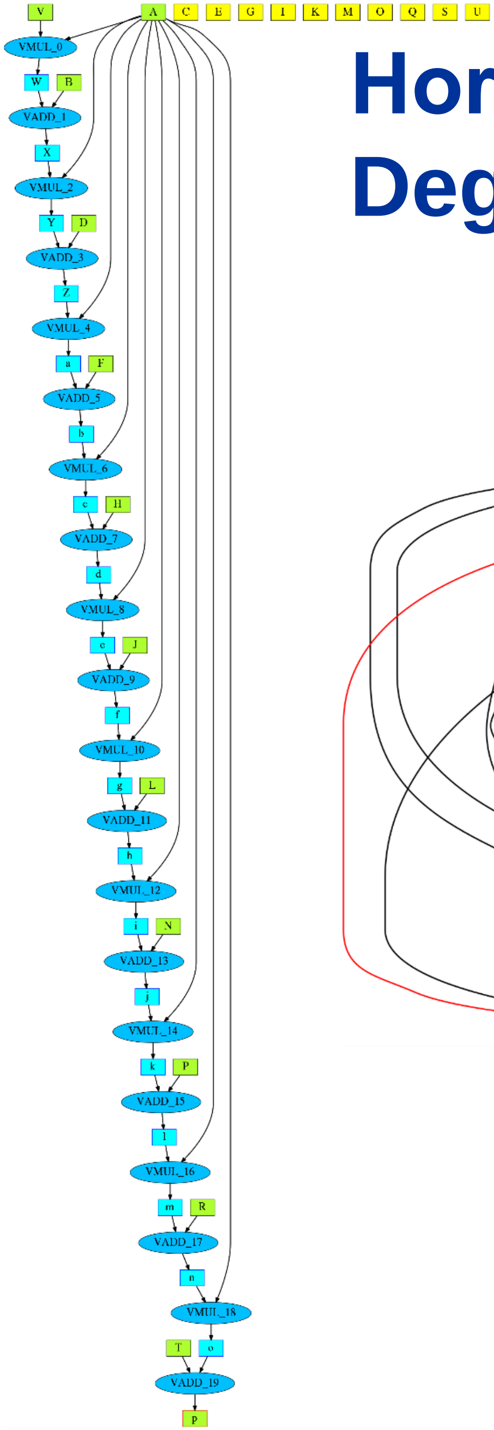


Evaluation of # Saved Transfer Startups and Optimization Time, cont.

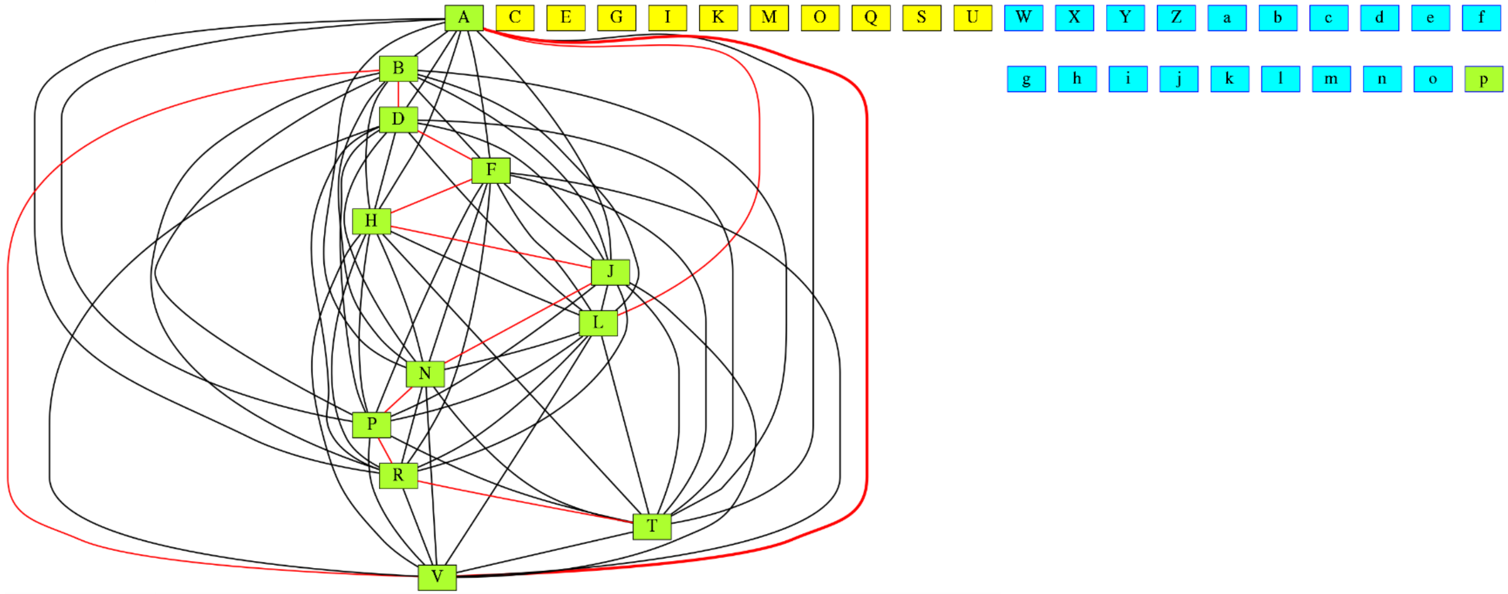
Special task graph topologies and application-derived task graphs

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

Horner’s Polynomial Evaluation, Degree 10 (N=20)

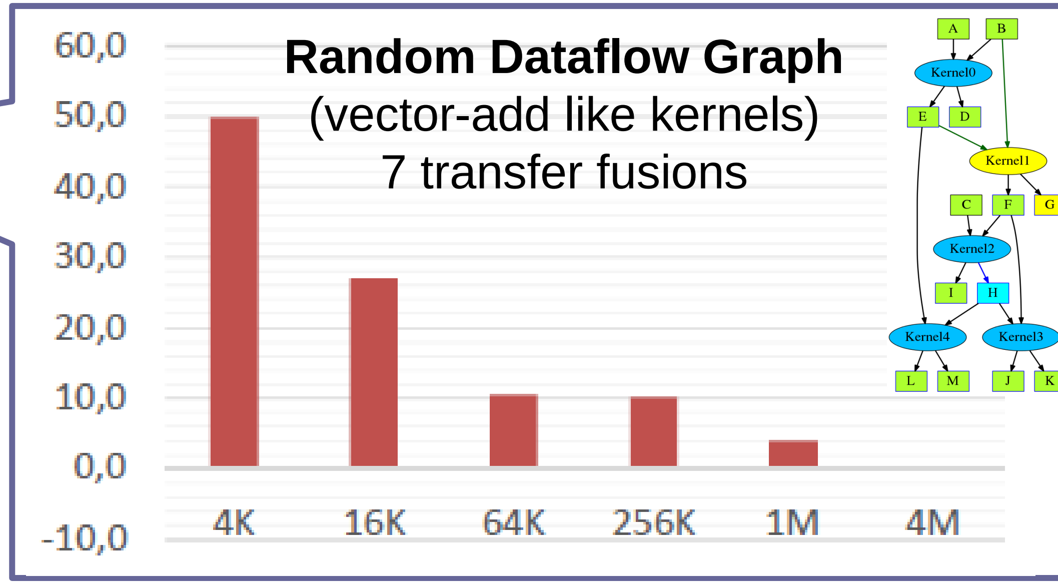


Affinity Graph:



Evaluation of Generated Transfer-Fusion Optimized CUDA Code

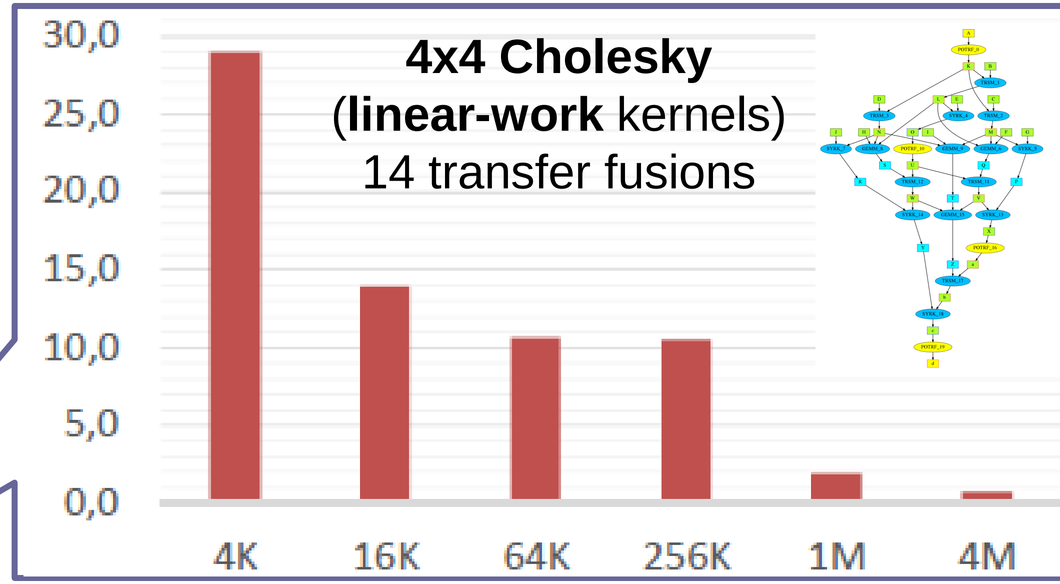
Vector Length [floats]	No Transfer Fusion [μs]	Transfer Fusion [μs]	Speedup [%]
Random Dataflow Graph ($N = 5$, $M_{init} = 3$)			
4K	211	141	49.6%
16K	415	327	26.9%
64K	1136	1028	10.5%
256K	4010	3641	10.1%
1M	14455	13907	3.9%
4M	55421	55464	-0.1%
Random Dataflow Graph ($N = 24$, $M_{init} = 10$)			
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%



- Allocation time not included
- Speedups between 5% and 170% at 4K float vectors
- Speedup decreases with operand size, but remains significant until 1M floats for kernels with low arithmetic intensity (e.g. vector-add, saxpy)
- Speedup quickly drops off for ≥64K floats for computation-heavy kernels, as transfer time gets insignificant (e.g. sgemm in Cholesky and MatMul)

Evaluation of Generated Transfer-Fusion Optimized CUDA Code

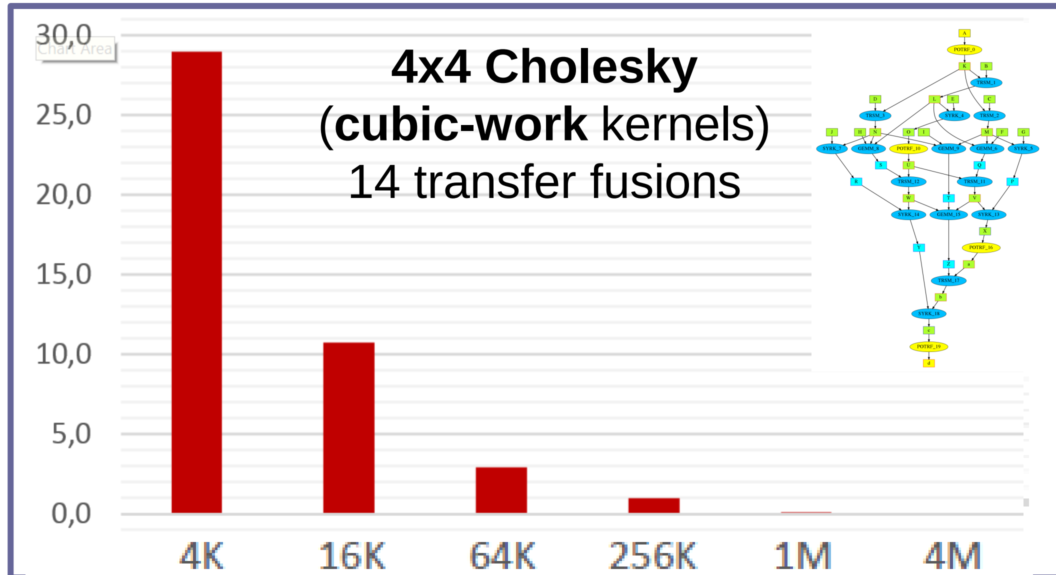
Vector Length [floats]	No Transfer Fusion [μs]	Transfer Fusion [μs]	Speedup [%]
Random Dataflow Graph ($N = 5$, $M_{init} = 3$)			
4K	211	141	49.6%
16K	415	327	26.9%
64K	1136	1028	10.5%
256K	4010	3641	10.1%
1M	14455	13907	3.9%
4M	55421	55464	-0.1%
Random Dataflow Graph ($N = 24$, $M_{init} = 10$)			
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%



- Allocation time not included
- Speedups between 5% and 170% at 4K float vectors
- Speedup decreases with operand size, but remains significant until 1M floats for kernels with low arithmetic intensity (e.g. vector-add, saxpy)
- Speedup quickly drops off for ≥64K floats for computation-heavy kernels, as transfer time gets insignificant (e.g. sgemm in Cholesky and MatMul)

Evaluation of Generated Transfer-Fusion Optimized CUDA Code

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%



- Allocation time not included
- Speedups between 5% and 170% at 4K float vectors
- Speedup decreases with operand size,
  but remains significant until 1M floats
  for kernels with low arithmetic intensity
  (e.g. vector-add, saxpy)
- Speedup quickly drops off for ≥64K floats for
  computation-heavy kernels,
  as transfer time gets insignificant
  (e.g. sgemm in Cholesky and MatMul)

CUDA code, including Fused Memory 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	[illegible]	4865	133.8%
1M	29229	14889	96.3%
[illegible]	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%

Due to the memory allocation fusion anomaly on our GPU at 2-operand-fusions

Summary and Outlook

- **Global-scope reordering** of vector variables in memory to optimize operand upload / download message fusion in distributed heterogeneous systems
  1. Analyze the Kernel-Vector Data-Flow Graph
  2. Build the Affinity Graph
  3. Max-weight Hamiltonian Paths in Affinity graph CCs
  4. Emitting code: Greedy vs. Tight strategy
- **Experiments** for synthetic kernel graphs
  - Low optimization time, good # transfer-fusions
  - Decent speedups for up to 1M elements and kernels of low oper. intensity
  - Prototype source code: www.ida.liu.se/~chrke/transferfusion
  - Future work: Tuning to decide up to which vector sizes to apply; work around the allocation fusion anomaly; use of optimized kernels
- **Possible usage scenarios**
  - *Static* optimization in kernel compilers (e.g., DSL)
  - *Dynamic* optimization: lazy execution builds runtime graph
    - Amortize optimization time over multiple iterations over same graph
- **Future extensions**: Combine with scheduling (and more...)

Questions?

References

This paper:

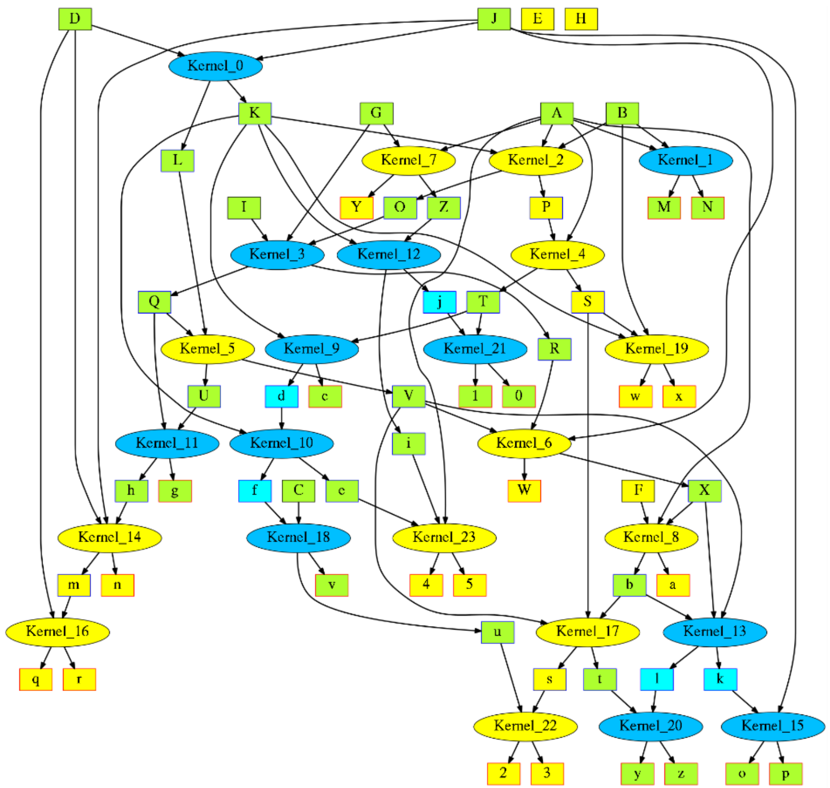
- Christoph Kessler: **Global optimization of operand transfer fusion in heterogeneous computing**. Proc. 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES-2019), St. Goar, Germany, May 2019. ACM. DOI: 10.1145/3323439.3323981

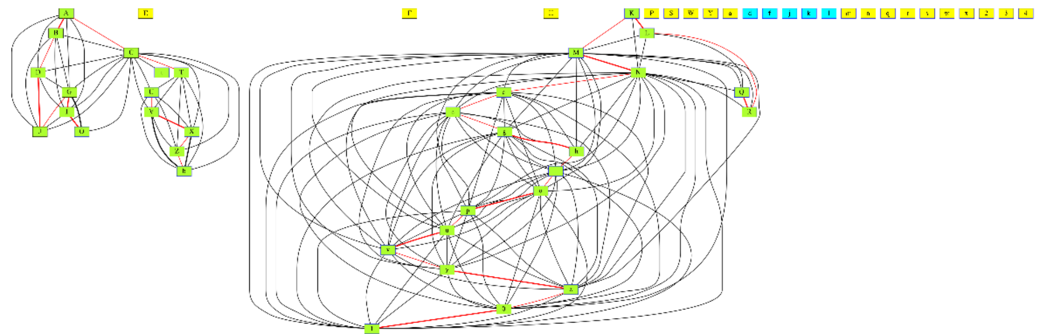
- Lu Li, Christoph Kessler: **Lazy Allocation and Transfer Fusion Optimization for GPU-based Heterogeneous Systems**. Proc. Euromicro PDP-2018 Int. Conf. on Parallel, Distributed, and Network-Based Processing, Cambridge, UK, Mar. 2018, IEEE.

- Prototype implementation source code: https://www.ida.liu.se/~chrke55/transferfusion/

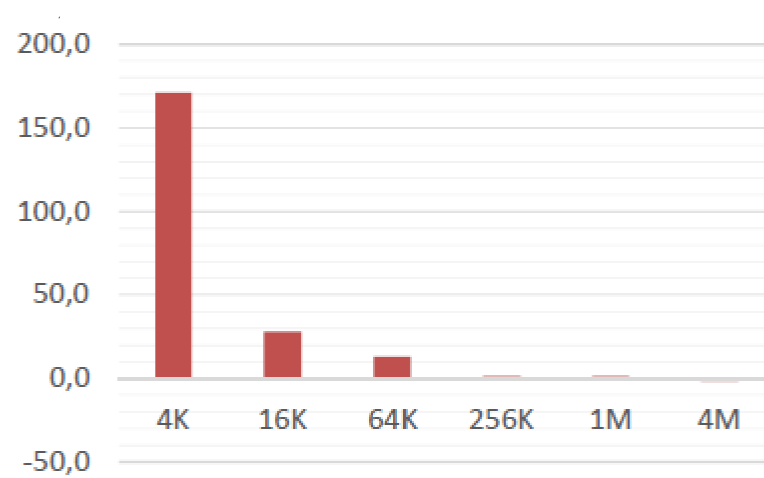
BACKUP SLIDES

Random DAG, N=24

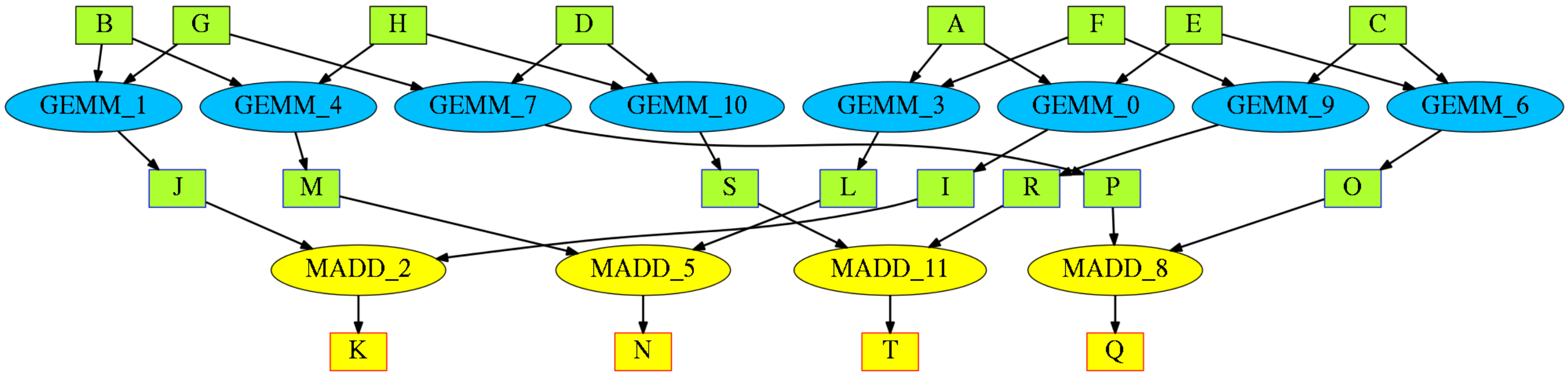


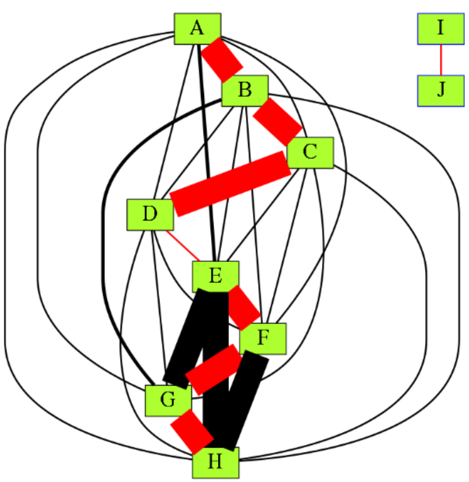


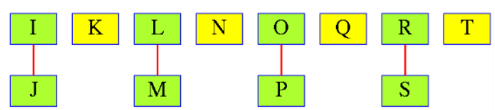
25 startups saved



2x2 Block Matrix-Matrix Multiply

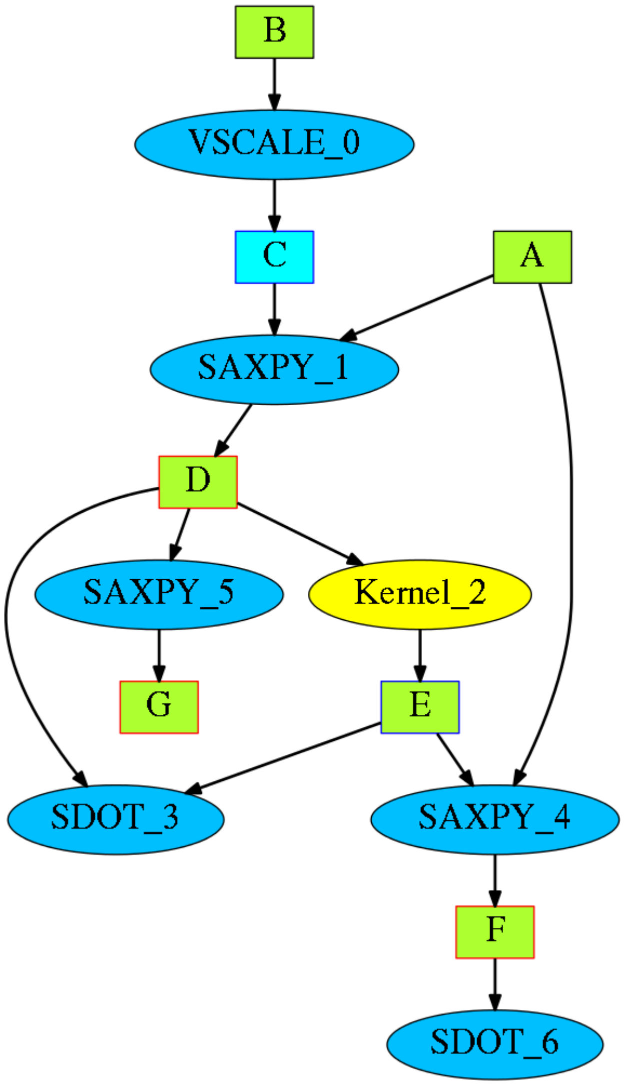


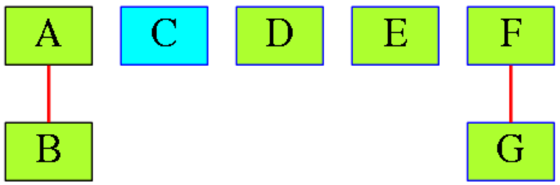




Gain: 11 startups

Conjugate Gradient Solver, Main Loop





Gain: 2 startups