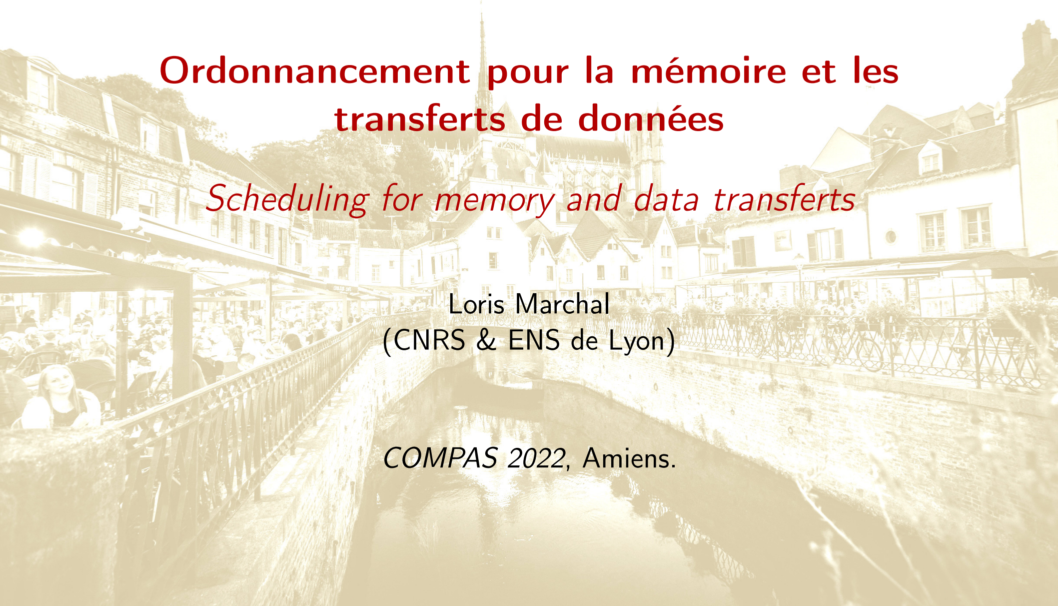


Ordonnancement pour la mémoire et les transferts de données

Scheduling for memory and data transferts

Loris Marchal
(CNRS & ENS de Lyon)

COMPAS 2022, Amiens.

Introduction & Motivation

- (Fast) Memory: place to store data for compute
- Always been a limited resource (4KB in Apollo 11 computer)
- Not limited anymore ?
  a few GB (laptops) – 1TB (servers)

- But problem size always gets bigger. . .

- . . . And this is rather a question of speed!
- Annual improvements:
  - Time per flop (computation): 59%
  - Data movement:

	Bandwidth	Latency
Network	26%	15%
DRAM	23%	5%

Numbers from *Getting up to speed: The future of supercomputing*, 2005, National Academies Press (2004 figure based on data on the period 1988-2002)

Introduction & Motivation

- (Fast) Memory: place to store data for compute
- Always been a limited resource (4KB in Apollo 11 computer)
- Not limited anymore ?
  a few GB (laptops) – 1TB (servers)
- But problem size always gets bigger. . .
- . . . And this is rather a question of speed!
- Annual improvements:
  - Time per flop (computation): 59%
  - Data movement:

	Bandwidth	Latency
Network	26%	15%
DRAM	23%	5%

Numbers from *Getting up to speed: The future of supercomputing*, 2005, National Academies Press (2004 figure based on data on the period 1988-2002)

Introduction & Motivation

- (Fast) Memory: place to store data for compute
- Always been a limited resource (4KB in Apollo 11 computer)
- Not limited anymore ?
  a few GB (laptops) – 1TB (servers)

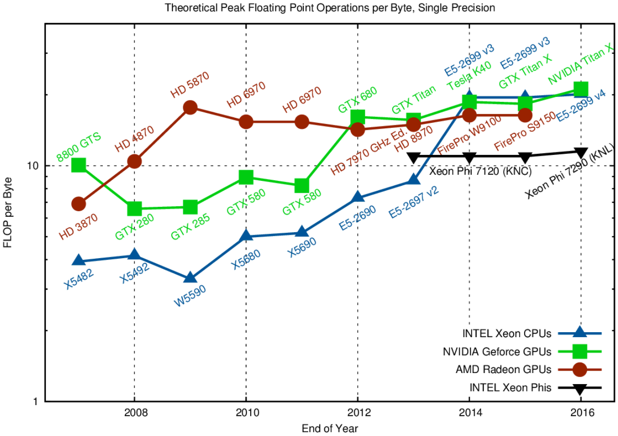
- But problem size always gets bigger…

- …And this is rather a question of speed!
- Annual improvements:
  - Time per flop (computation): 59%
  - Data movement:

	Bandwidth	Latency
Network	26%	15%
DRAM	23%	5%

Numbers from *Getting up to speed: The future of supercomputing*, 2005, National Academies Press (2004 figure based on data on the period 1988-2002)

Flop per byte moved ratio

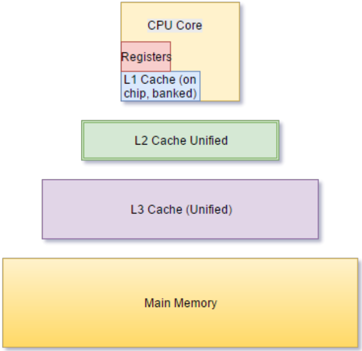


ratio computing speed/communication speed

From `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`

Beyond the memory wall

- Time to move the data $>$ Time to compute on the data
- Similar problem in microprocessor design: "memory wall"
- Traditional workaround:
  add a faster but smaller "cache" memory
- Now a hierarchy of caches !



Computing with bounded cache/memory

- Limited amount of fast cache
- Performance sensitive to data locality
- Optimize data reuse
- Avoid data movements between memory and cache(s) (time-consuming and energy-consuming)

In this talk: some algorithmic approaches to this problem

Computing with bounded cache/memory

- Limited amount of fast cache
- Performance sensitive to data locality
- Optimize data reuse
- Avoid data movements between memory and cache(s) (time-consuming and energy-consuming)

In this talk: some algorithmic approaches to this problem

Outline

Outline

Impact of Algorithm Design on Data Movements

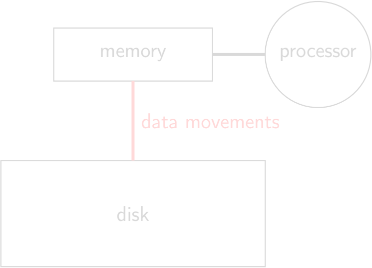
Scheduling Task Graphs with Limited Memory

Reducing Data Movements for Independant Tasks on GPUs

Example: matrix-matrix product

- Consider two square matrices A and B (size $n \times n$)
- Compute generalized matrix product: $C \leftarrow C + AB$

```
Simple-Matrix-Multiply($n, C, A, B$)
for $i = 0 \rightarrow n-1$ do
    for $j = 0 \rightarrow n-1$ do
        for $k = 0 \rightarrow n-1$ do
            $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```



Assume simple two-level memory model:

- Slow but infinite disk storage
  (where A and B are originally stored)
- Fast and limited memory (size M)

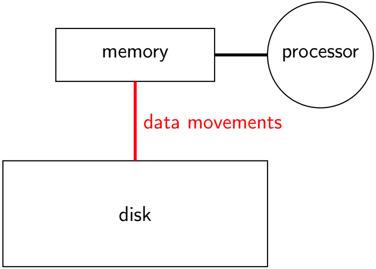
Objective: limit data movement between disk/memory

NB: also applies to other two-level systems (memory/cache, etc.)

Example: matrix-matrix product

- Consider two square matrices A and B (size $n \times n$)
- Compute generalized matrix product: $C \leftarrow C + AB$

```
Simple-Matrix-Multiply($n, C, A, B$)
for $i = 0 \rightarrow n-1$ do
    for $j = 0 \rightarrow n-1$ do
        for $k = 0 \rightarrow n-1$ do
            $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```



Assume simple two-level memory model:

- Slow but infinite disk storage (where A and B are originally stored)
- Fast and limited memory (size M)

Objective: limit data movement between disk/memory

NB: also applies to other two-level systems (memory/cache, etc.)

Simple algorithm analysis

```
Simple-Matrix-Multiply($n, C, A, B$)
for $i = 0 \to n-1$ do
    for $j = 0 \to n-1$ do
        for $k = 0 \to n-1$ do
            $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```

- Assume the memory cannot store half of a matrix: $M < n^2/2$
- Question: How many data movement in this algorithm ?

Simple algorithm analysis

```
Simple-Matrix-Multiply($n, C, A, B$)
for $i = 0 \to n-1$ do
    for $j = 0 \to n-1$ do
        for $k = 0 \to n-1$ do
            $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```

- Assume the memory cannot store half of a matrix: $M < n^2/2$
- Question: How many data movement in this algorithm ?

Answer:

- all elements of B accessed during one iteration of the outer loop
- At most half of B stays in memory
- At least $n^2/2$ elements must be read per outer loop
- At least $n^3/2$ read for entire algorithms
- Same order of magnitude of computations: $O(n^3)$
- Very bad data reuse ☹ Question: How to do better ?

Blocked matrix-matrix product

- Divide each matrix into blocks of size $b \times b$:
  $A^b_{i,k}$ is the block of A at position (i, k)
- Perform "coarse-grain" matrix product on blocks
- Perform each block product with previous algorithms

```
Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ do
    for $j = 0, \rightarrow n/b - 1$ do
        for $k = 0, \rightarrow n/b - 1$ do
            Simple-Matrix-Multiply($n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j}$)
```

Blocked matrix-matrix product – Analysis

```
Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ do
    for $j = 0, \rightarrow n/b - 1$ do
        for $k = 0, \rightarrow n/b - 1$ do
            Simple-Matrix-Multiply($n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j}$)
```

Question: Number of data movements ?

Blocked matrix-matrix product – Analysis

```
Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ do
    for $j = 0, \rightarrow n/b - 1$ do
        for $k = 0, \rightarrow n/b - 1$ do
            Simple-Matrix-Multiply($n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j}$)
```

Question: Number of data movements ?

- Iteration of inner loop: 3 blocks of size $b \times b = \sqrt{M/3}^3 = M/3$ → fits in memory
- At most $M + M/3$ ($O(M)$) data movements for each inner loop (reading/writing)
- Number of inner iterations: $(n/b)^3 = n^3/(M/3) = O(n^3/M\sqrt{M})$
- Total number of data movements: $O(n^3/\sqrt{M})$

Blocked matrix-matrix product – Analysis

```
Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ do
    for $j = 0, \rightarrow n/b - 1$ do
        for $k = 0, \rightarrow n/b - 1$ do
            Simple-Matrix-Multiply($n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j}$)
```

Question: Number of data movements ?

- Iteration of inner loop: 3 blocks of size $b \times b = \sqrt{M/3}^3 = M/3$ → fits in memory
- At most $M + M/3$ ($O(M)$) data movements for each inner loop (reading/writing)
- Number of inner iterations: $(n/b)^3 = n^3/(M/3) = O(n^3/M\sqrt{M})$
- Total number of data movements: $O(n^3/\sqrt{M})$

Question: Can we do (significantly) better ?

Decompose the Computation into Phases

- Phase: consecutive subsequence of computation with exactly M read operations
  (last phase may have $< M$ reads)
- Number of data available for computation in each phase:

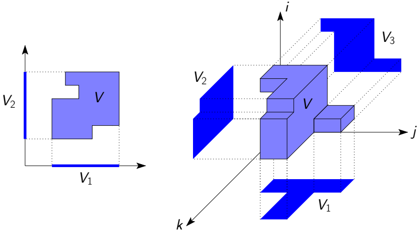
  $$\leq M \text{ (initially in the memory) } + M \text{ (read during the phase)}$$

- Crude bound on the number of elements of A, B and C used/computed: $N_A \leq 2M,\ N_B \leq 2M, N_C \leq 2M$

A Helpful Geometric Lemma

Theorem (Irony, Toledo, Tiskin, 2008).

Using N_A elements of A, N_B elements of B and N_C elements of C, we can perform at most $\sqrt{N_A N_B N_C}$ distinct products.



Theorem (Discrete Loomis-Whitney Inequality).

Let V be a finite subset of $\mathbb{Z}^3$ and V_1, V_2, V_3 denotes the orthogonal projections of V on each coordinate planes, we have

$$|V|^2 \leq |V_1| \cdot |V_2| \cdot |V_3|,$$

I/0 Bound

- Number of elementary product done in one phase: at most $\sqrt{N_A N_B N_C} \leq \sqrt{(2M)^3}$
- In total: n^3 elementary products
- Number of phases: at least $\dfrac{n^3}{\sqrt{(2M)^3}}$
- Number of reads: at least $\dfrac{n^3}{2\sqrt{2}\sqrt{M}}$

Theorem (refined version, Langou 2019).

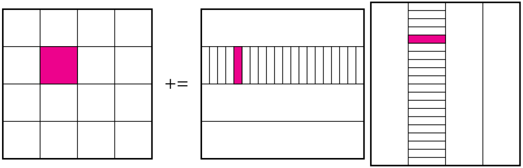
The total volume of I/Os is bounded by:

$$V_{I/O} \geq \frac{2N^3}{\sqrt{M}} + N^2 - 2M$$

Optimal algorithm

Consider the following algorithm sketch:

- Partition C into blocks of size $(\sqrt{M}-1)\times(\sqrt{M}-1)$
- Partition A into block-columns of size $(\sqrt{M}-1)\times 1$
- Partition B into block-rows of size $1\times(\sqrt{M}-1)$
- For each block C_b of C:
    - Load the corresponding blocks of A and B on after the other
    - For each pair of blocks A_b, B_b, compute $C_b \leftarrow C_b + A_b B_b$
    - When all products for C_b are performed, write back C_b

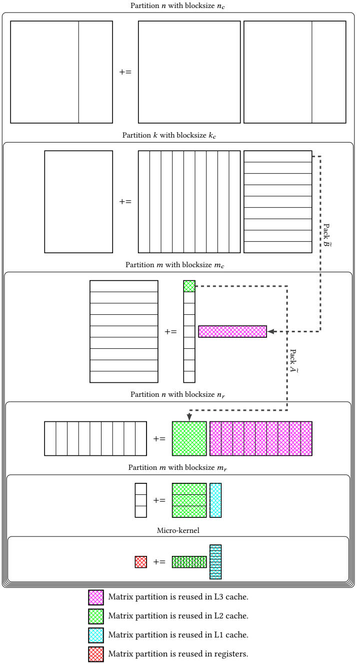


Purely theoretical algorithm?

BLAS: Basic Linear Algebra Subprograms

- Introduced in the 80s as a standard for LA computations
- Written first in FORTRAN
- Library provided by the vendor to ease use of new machines
- Automatic Tuning: ATLAS
- GotoBLAS

Matrix product: still a large share of LA computations



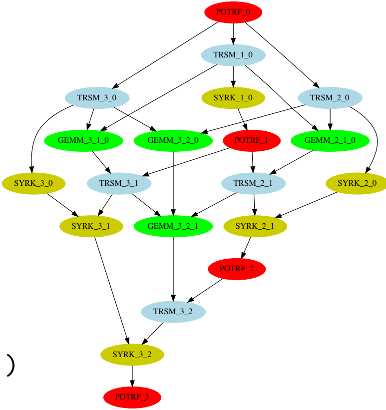
Outline

Impact of Algorithm Design on Data Movements

Scheduling Task Graphs with Limited Memory

Reducing Data Movements for Independant Tasks on GPUs

Taming HPC platforms with runtime systems



- Write you application as function calls (tasks),
- Specify data input/output (dependencies)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!

```
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0; k<N; k++)
         MULT_ADD(C[i,j], A[i,k], B[k,j])
```

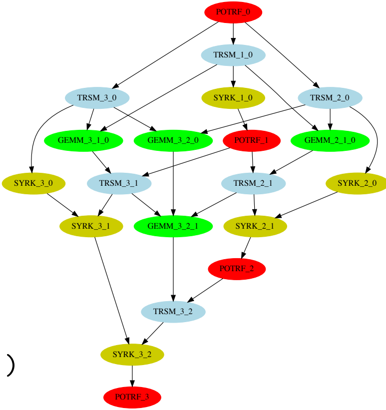
Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Taming HPC platforms with runtime systems

- Write you application as function calls (tasks),
- Specify data input/output (dependencies)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!

```
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0; k<N; k++)
         MULT_ADD(C[i,j], A[i,k], B[k,j])
```

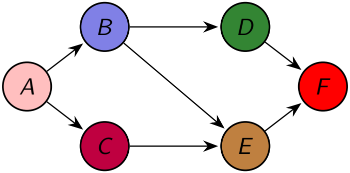


Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Task graph scheduling and memory

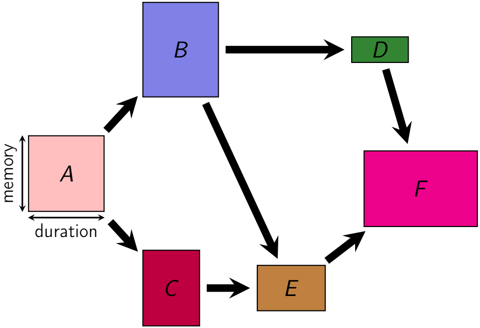
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

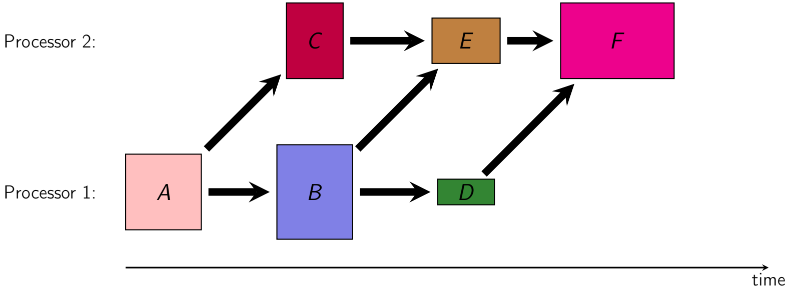
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

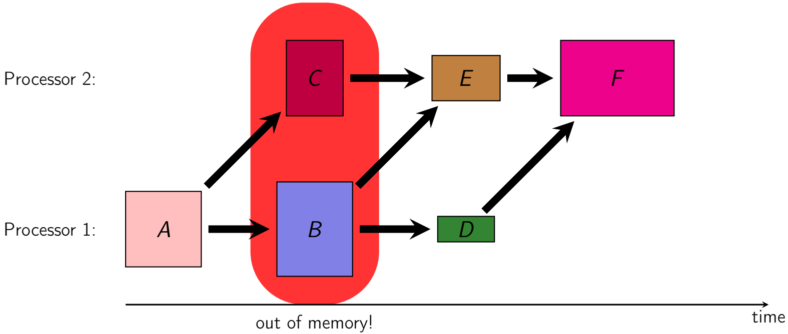
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

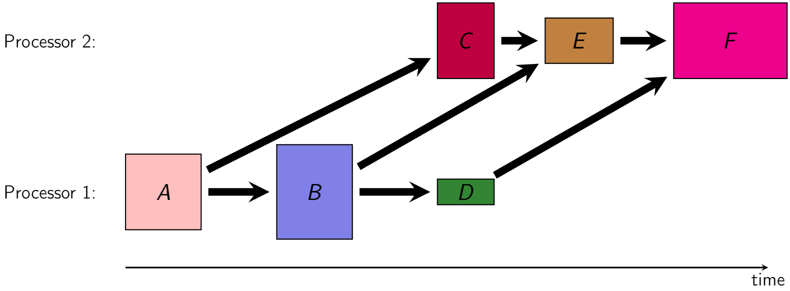
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

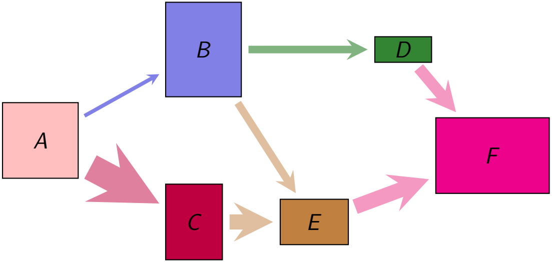
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

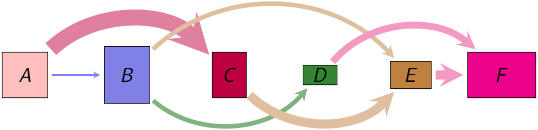
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



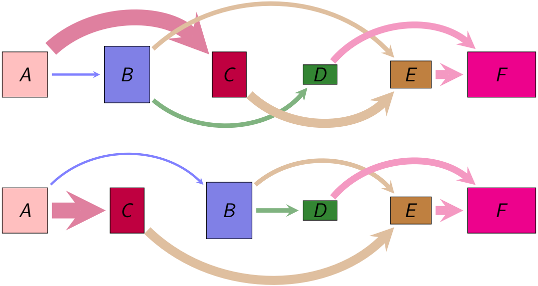
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



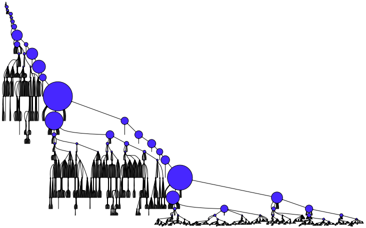
Going back to sequential processing

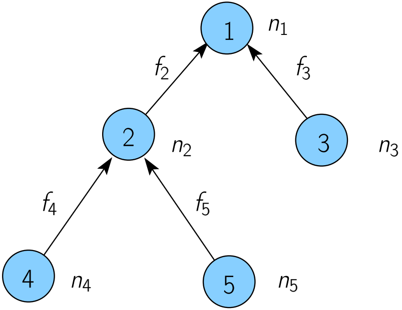
- Temporary data require memory
- Scheduling influences the peak memory



Tree-shaped task graphs

- Multifrontal sparse matrix factorization over runtimes
- Task graph: tree (with dependencies towards the root)
- Large temporary data



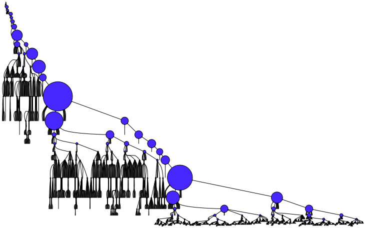


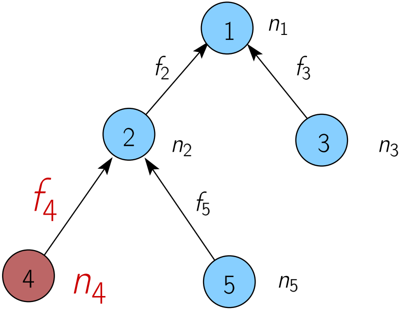
- Output data of size f_i
- Execution data of size n_i
- Memory for processing node i:

$$\left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$$

Tree-shaped task graphs

- Multifrontal sparse matrix factorization over runtimes
- Task graph: tree (with dependencies towards the root)
- Large temporary data



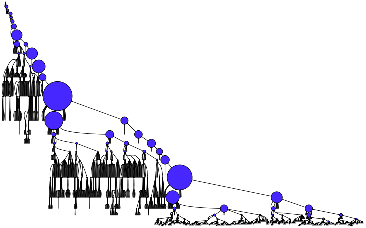


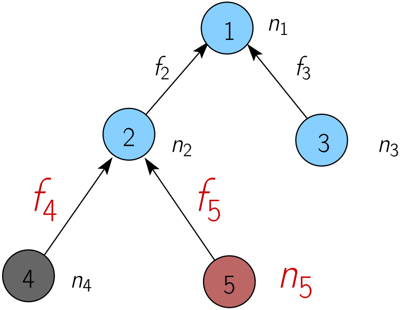
- Output data of size f_i
- Execution data of size n_i
- Memory for processing node i:

$$\left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$$

Tree-shaped task graphs

- Multifrontal sparse matrix factorization over runtimes
- Task graph: tree (with dependencies towards the root)
- Large temporary data



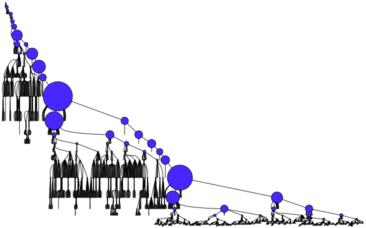


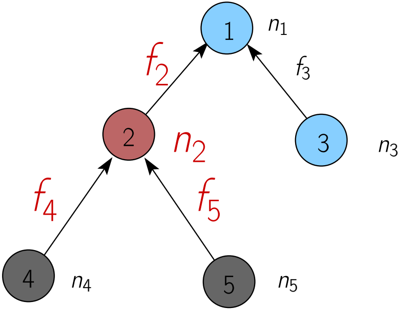
- Output data of size f_i
- Execution data of size n_i
- Memory for processing node i:

$$\left(\sum_{j\in Children(i)} f_j\right) + n_i + f_i$$

Tree-shaped task graphs

- Multifrontal sparse matrix factorization over runtimes
- Task graph: tree (with dependencies towards the root)
- Large temporary data



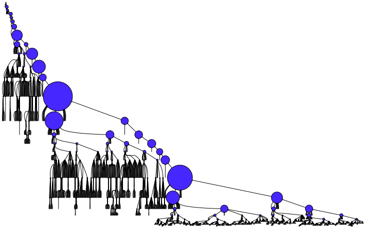


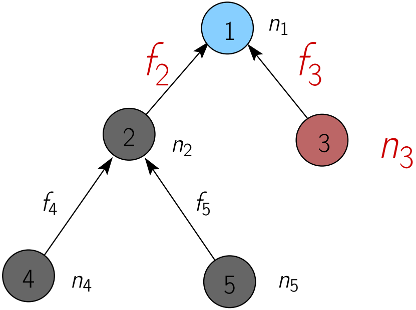
- Output data of size f_i
- Execution data of size n_i
- Memory for processing node i:

$$\left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$$

Tree-shaped task graphs

- Multifrontal sparse matrix factorization over runtimes
- Task graph: tree (with dependencies towards the root)
- Large temporary data



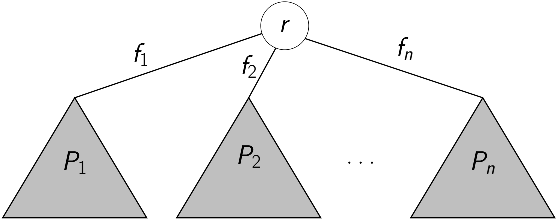


- Output data of size f_i
- Execution data of size n_i
- Memory for processing node i:

$$\left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$$

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



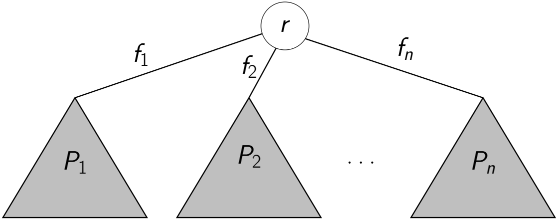
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



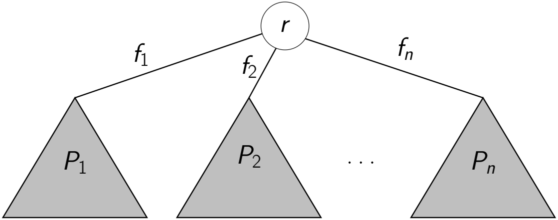
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \ldots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



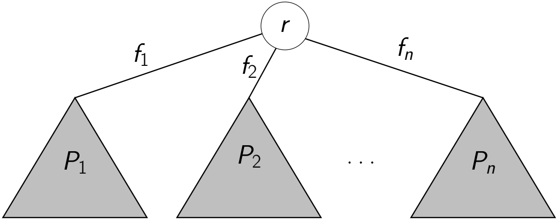
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



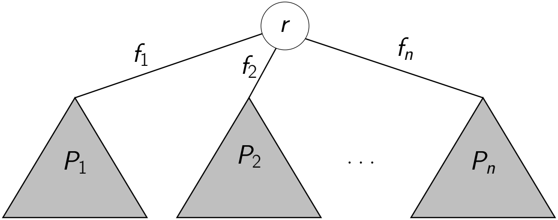
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



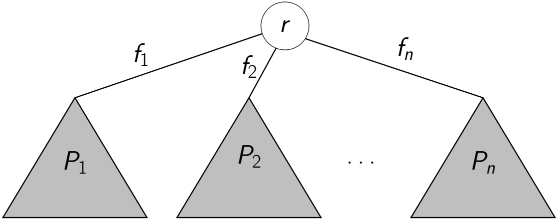
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:

Liu's best post-order traversal for trees

Post-Order: entirely process one subtree after the other (DFS)



- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order: non-increasing $P_i - f_i$

Results on task graph scheduling

Minimize the memory with single processor:

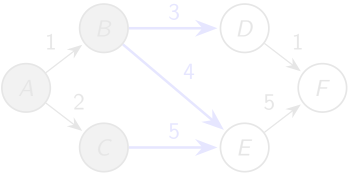
- ☺ Best post-order schedule on trees [Liu 1986]
- ☺ Optimal schedule on trees [Liu 1987]
- ☺ Optimal schedule for Series-Parallel Graphs [Kayaaslan et al., 2018]
- ☹ General graphs: PSPACE complete [Gilbert et al., 1980]

Parallel processing: bi-criteria problem (makespan and shared memory)

- ☹ NP-complete on trees [Marchal et al., 2013]

What to do if you cannot control the schedule?

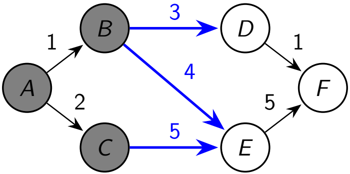
- Task graph scheduled at runtime (dynamic)
- How to make sure not to exceed available memory?



1. Compute worse achievable memory
   (topological cut with maximum weight)
2. If needed, add new dependencies to prevent this worse situation

What to do if you cannot control the schedule?

- Task graph scheduled at runtime (dynamic)
- How to make sure not to exceed available memory?



1. Compute worse achievable memory
   (topological cut with maximum weight)
2. If needed, add new dependencies to prevent this worse situation

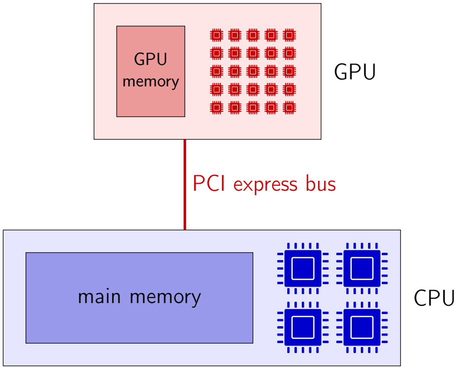
Outline

Impact of Algorithm Design on Data Movements

Scheduling Task Graphs with Limited Memory

Reducing Data Movements for Independant Tasks on GPUs

Platform model



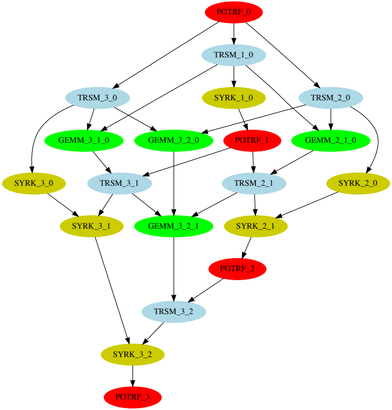
GPUs provide large speed-ups for reduced energy, but:

- limited memory within GPU
- connected through bus with limited bandwidth

Tasks available for scheduling in runtime systems

At any time step: consider only available tasks

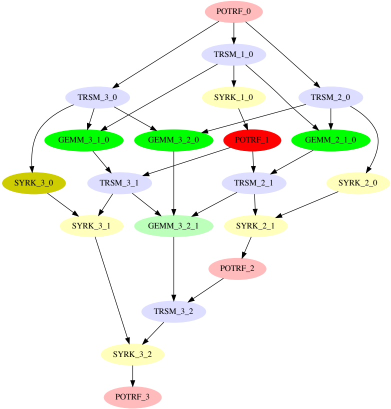
- Independant tasks (no dependency among tasks)
- Sharing some input data



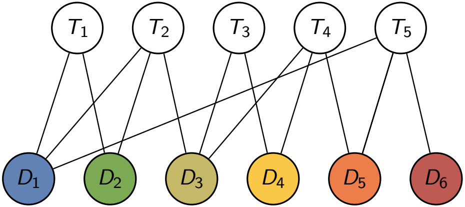
Tasks available for scheduling in runtime systems

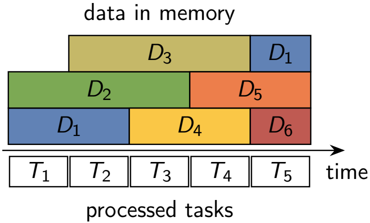
At any time step: consider only available tasks

- Independant tasks (no dependency among tasks)
- Sharing some input data



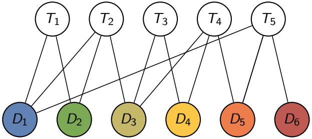
Independant tasks sharing data





- Bipartite graph modeling data sharing among tasks
- Only 3 data allowed in memory (in this example)
- Some data may be evicted/reloaded (D_1 here)

Problem modeling



- Bipartite graph (tasks sharing input data)
- Homogeneous data (size=1)
- Homogeneous tasks (duration=1)
- Limited memory M

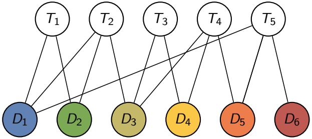
Objective: minimize data loads

Execution framework: repeat these 3 phases

1. Evict some data from the memory
2. Load some new data
3. Compute next task

A priori: complex description of the solution

Problem modeling



- Bipartite graph (tasks sharing input data)
- Homogeneous data (size=1)
- Homogeneous tasks (duration=1)
- Limited memory M

Objective: minimize data loads

Execution framework: repeat these 3 phases

1. Evict some data from the memory → which data to evict?
2. Load some new data → which data to load?
3. Compute next task → which task order?

A priori: complex description of the solution

Simplifying the solution

Say we decided the task order.

Theorem (straightforward).

Thou shalt load data as late as possible.

⇒ Load (missing) data for a task right before its processing.

Theorem (adaptation of Belady's rule).

Thou shalt evict data whose next usage is the furthest in the future.

Belady's rule: optimal policy for cache management

- Difference: here each task requests several data

So we only need to compute the best task order!

Simplifying the solution

Say we decided the task order.

Theorem (straightforward).

Thou shalt load data as late as possible.

$\Rightarrow$ Load (missing) data for a task right before its processing.

Theorem (adaptation of Belady's rule).

Thou shalt evict data whose next usage is the furthest in the future.

Belady's rule: optimal policy for cache management

- Difference: here each task requests several data

So we only need to compute the best task order!

Simplifying the solution

Say we decided the task order.

Theorem (straightforward).

Thou shalt load data as late as possible.

⇒ Load (missing) data for a task right before its processing.

Theorem (adaptation of Belady's rule).

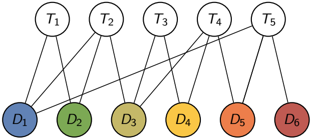
Thou shalt evict data whose next usage is the furthest in the future.

Belady's rule: optimal policy for cache management

- Difference: here each task requests several data

So we only need to compute the best task order!

Back to our problem



- Tasks sharing input data
- Limited memory M
- Objective: minimize data loads

Repeat:

1. If needed, evict data used furthest in the future
2. Load missing data for next task
3. Compute next task

Until all tasks are processed.

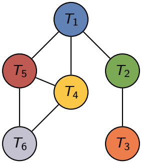
Single question: find task order

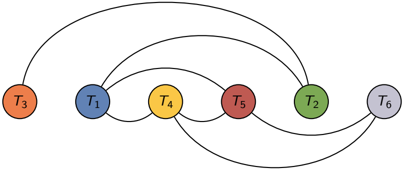
Link to cutwidth minimization

Special case:

- Each data shared by at most 2 tasks
- Objective: Load each data exactly once (never evict useful data)

Another graph model: vertices=tasks, edges=data shared among tasks





- Ordering tasks ⇔ Linear arrangement of vertices
- Amount of data in memory ⇔ cutwidth
  (maximum number of edges cut by a vertical line)

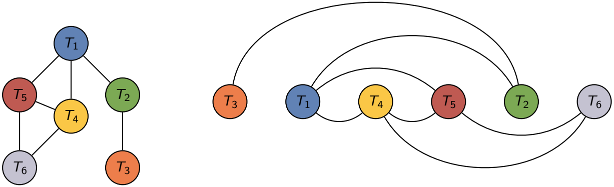
Our problem is NP-complete by reduction to Cutwidth Minimization.

Link to cutwidth minimization

Special case:

- Each data shared by at most 2 tasks
- Objective: Load each data exactly once (never evict useful data)

Another graph model: vertices=tasks, edges=data shared among tasks



- Ordering tasks ⇔ Linear arrangement of vertices
- Amount of data in memory ⇔ cutwidth
  (maximum number of edges cut by a vertical line)

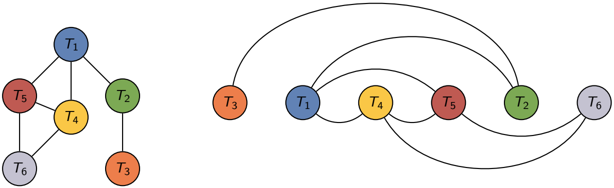
Our problem is NP-complete by reduction to Cutwidth Minimization.

Link to cutwidth minimization

Special case:

- Each data shared by at most 2 tasks
- Objective: Load each data exactly once (never evict useful data)

Another graph model: vertices=tasks, edges=data shared among tasks



- Ordering tasks ⇔ Linear arrangement of vertices
- Amount of data in memory ⇔ cutwidth
  (maximum number of edges cut by a vertical line)

Our problem is NP-complete by reduction to Cutwidth Minimization.

Building packages of tasks

When the problem is too hard

- Change the problem!

Build packages of tasks sharing a lot of common data

- All inputs within a package fit in memory
- Minimal number of packages

Then, schedule packages one after the others

Building packages of tasks

When the problem is too hard

- Change the problem!

Build packages of tasks sharing a lot of common data

- All inputs within a package fit in memory
- Minimal number of packages

Then, schedule packages one after the others

Unfortunately, this is also an NP-complete problem ☹

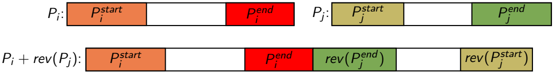
Heuristic to build packages

Hierarchical Fair Packing:

1. Start with each task being a package
2. Merge small packages sharing many input data
3. Stop when total input data exceed memory bound

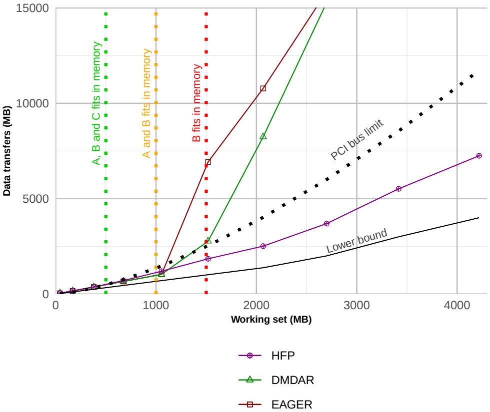
Optimizations:

- Package flipping:
  reverse some package to improve data reuse



- Continue merging packages when the memory bound is reached:
  improve locality among packages

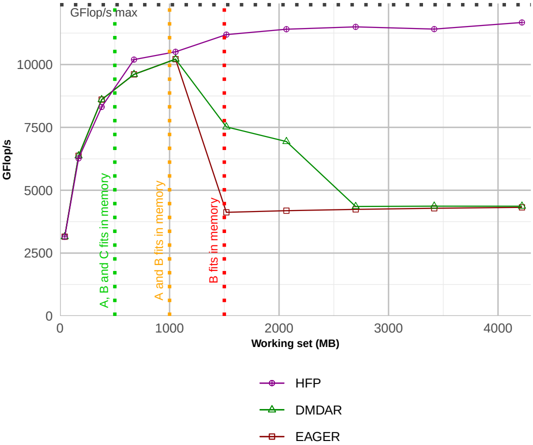
Validation – data movements



- Schedulers implemented in StarPU
- Main competitor: DMDAR (actual scheduler of StarPU)
  - (Allocate tasks to the resource that will complete it the earliest)
  - Reorder tasks at runtime to favor tasks with fewest load requests
- EAGER: follow submission order

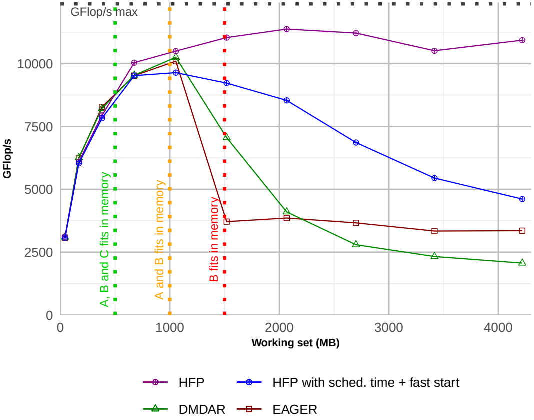
- 3D matrix multiplication
- Data-movements close to the lower bounds
- DMDAR leads to large data-movement as soon as memory is limited

Validation – performance in simulations



- Optimizing data movements allows to keep peak performance even when memory is limited

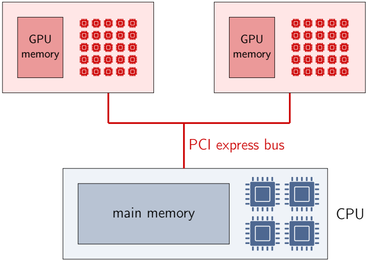
Validation – performance in real experiments



- Performance very similar to simulation for small sizes
- Impact of the complexity for large sizes

Shortcomings & final objective

- Large pre-computation time for large sizes
  (comparing and merging the packages)
- Real objective: distributed setting
  Several GPUs, with their own memory, sharing the bus



Two problems:

- Partition tasks among GPUs
- Order tasks within a GPU

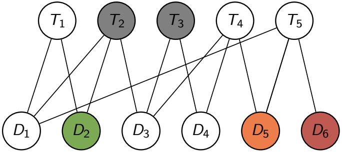
Demand-driven heuristics

Whenever a GPU requires some more work:

- Find the new data that enables the greatest number of available tasks
- Transfer this new data
- Allocate all enabled tasks to the GPU

What about eviction:

- No complete vision of the future ☹
- Window of allocated tasks ☺
- Perform Belady's rule with this limited prediction



DARTS (Data-Aware Reactive Task Scheduling)

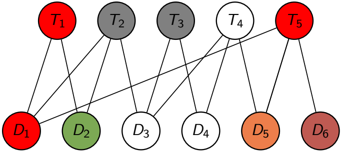
Demand-driven heuristics

Whenever a GPU requires some more work:

- Find the new data that enables the greatest number of available tasks
- Transfer this new data
- Allocate all enabled tasks to the GPU

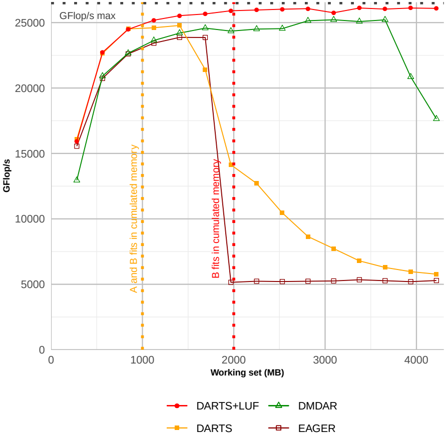
What about eviction:

- No complete vision of the future ☹
- Window of allocated tasks ☺
- Perform Belady's rule with this limited prediction



DARTS (Data-Aware Reactive Task Scheduling)

Performance on 2 GPUs (real experiments)



- DARTS is able to achieve peak performance
- Good eviction policy is critical !
  (LUF:adapted Belady's rule, otherwise:LRU)

Conclusion

Take-away messages:

- Concentrate on data movements is the key for performance
- Algorithm design can help re-organizing computations for better data reuse
- With help from: compilation, cache management, . . .
- Runtime scheduling of task graphs: avenue for scheduling research, with specific constraints
  (low complexity, limited knowledge, possible pre-computation, . . .)