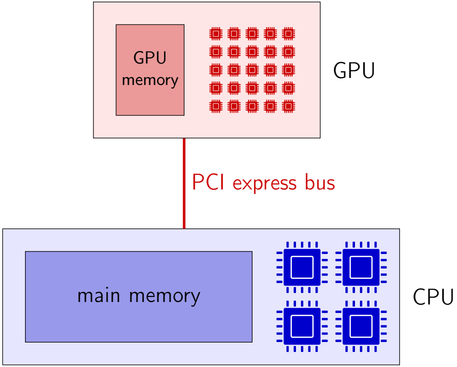
Scheduling of Tasks Sharing Data on GPUs with limited memory

Loris Marchal (CNRS & ENS-Lyon)
Joint work with Maxime Gonthier & Samuel Thibault (Inria Bordeaux)

New Challenges in Scheduling Theory, Aussois, May 2022.

Platform model



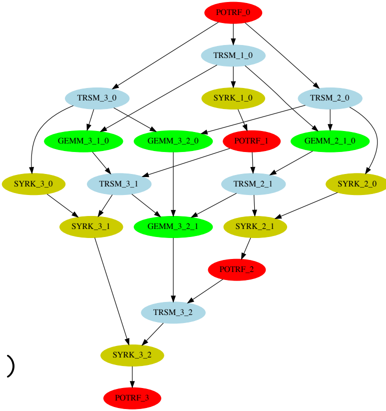
GPUs provide large speed-ups for reduced energy, but:

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

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])
```



At any time step: consider only available tasks

- Independant tasks
- Sharing some input data

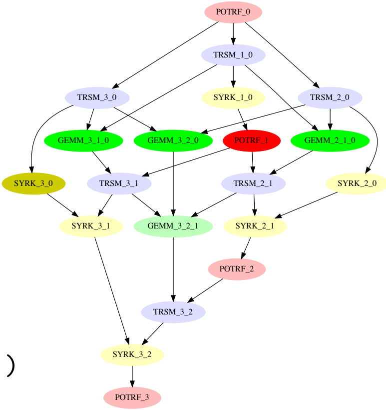
Taming HPC platforms with runtime systems

- 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!

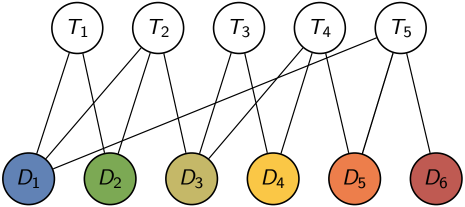
```
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0; k<N; k++)
         MULT_ADD(C[i,j], A[i,k], B[k,j])
```

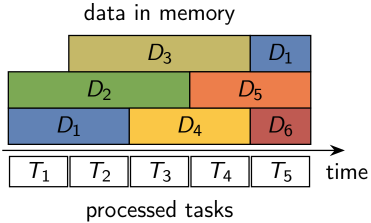
At any time step: consider only available tasks

- Independant tasks
- Sharing some input data



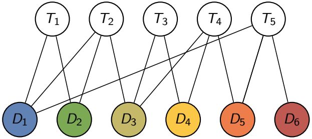
Independant tasks sharing data





- 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

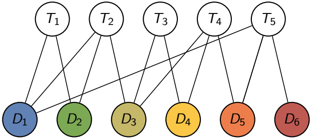
Objective: minimize data loads

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 $\rightarrow$ which data to evict?
2. Load some new data $\rightarrow$ which data to load?
3. Compute next task $\rightarrow$ 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.

$\Rightarrow$ 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.

$\Rightarrow$ 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.

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

Theorem (adaptation of Belady's rule).

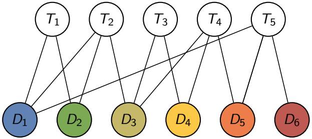
Thou shalt evict data whose next usage is the furthest in the future.

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!

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.

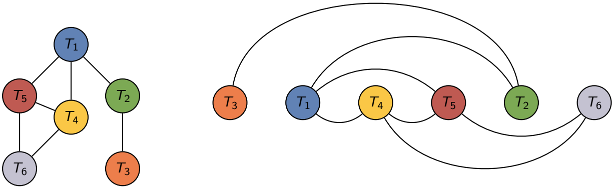
Single question: find task order

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 $\Leftrightarrow$ Linear arrangement of vertices
- Amount of data in memory $\Leftrightarrow$ cutwidth
  (maximum number of edges cut by a vertical line)

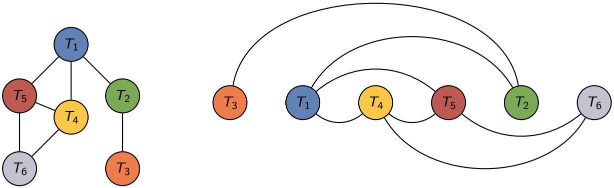
Our problem is NP-complete by reduction to Cutwidth Minimization.

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 $\Leftrightarrow$ Linear arrangement of vertices
- Amount of data in memory $\Leftrightarrow$ cutwidth
  (maximum number of edges cut by a vertical line)

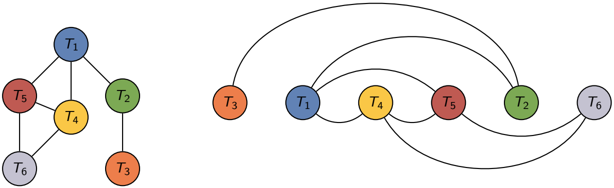
Our problem is NP-complete by reduction to Cutwidth Minimization.

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 $\Leftrightarrow$ Linear arrangement of vertices
- Amount of data in memory $\Leftrightarrow$ 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 ☹

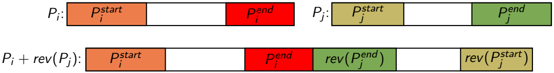
Heuristic to build packages

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

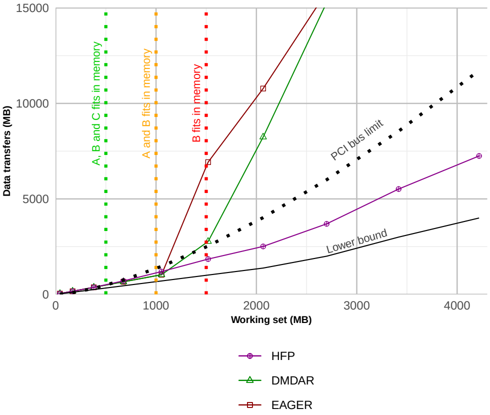
Optimizations:

- Package flipping:
  reverse some package to improve data reuse



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

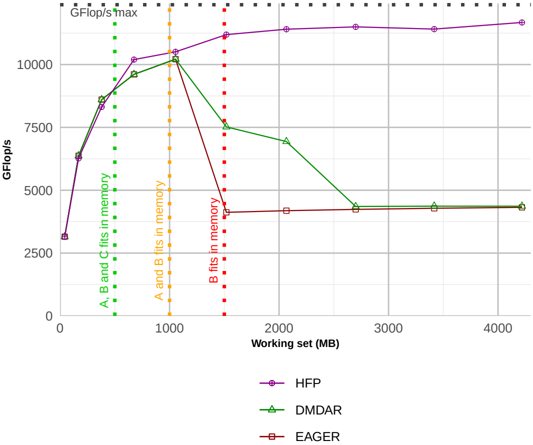
Validation – data movements



- 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

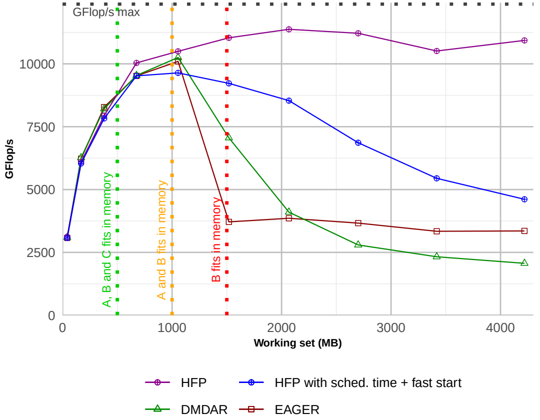
- 3D matrix multiplication
- Data-movements close to the lower bounds
- DMDAR leads to large data-movement as soon as memory is limited

Validation – performance in simulations



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

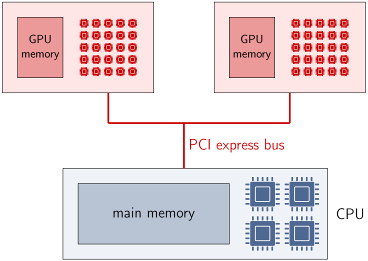
Validation – performance in real experiments



- 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

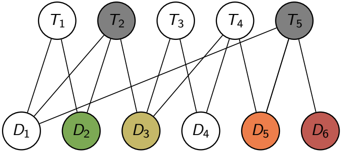
Demand-driven heuristics

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)

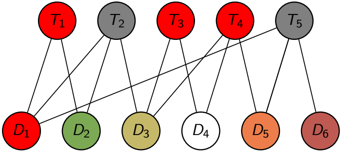
Demand-driven heuristics

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

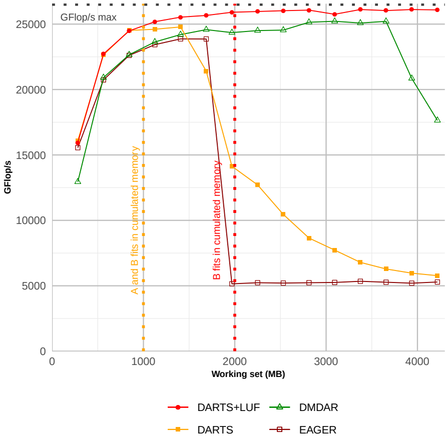
What about eviction:

- 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
- Runtime scheduling of task graphs → independant tasks at each step
- Need for very fast heuristics
  (pre-computation can be allowed)
- Cache management with good knowledge of future requests

Next step:

- Trade-off data locality vs. task affinity (CPU/GPU)