Tasks approach, StarPU

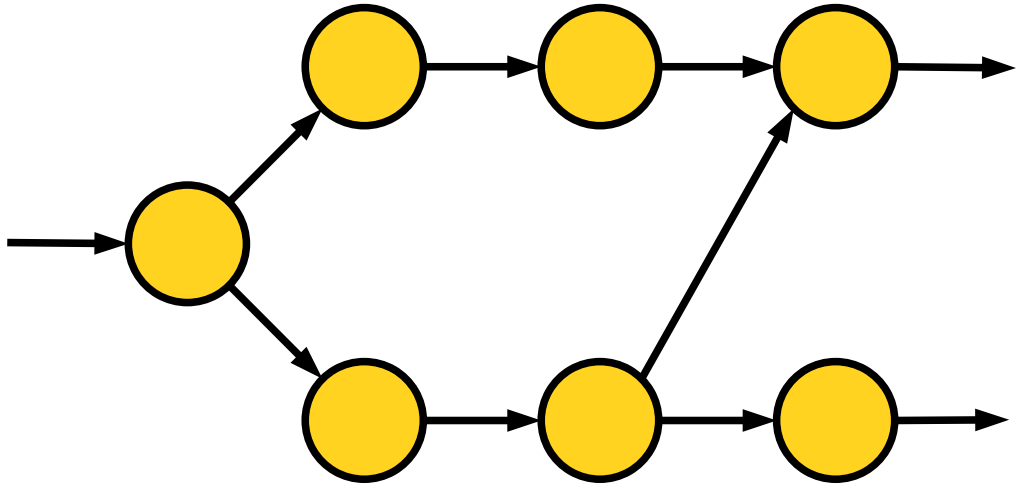
Samuel Thibault

INRIA Bordeaux - Sud-Ouest -- STORM Team

Task graphs

- Well-studied for scheduling parallelism (since 60's!)
- But only recent trend in HPC
- Departs from usual sequential programming

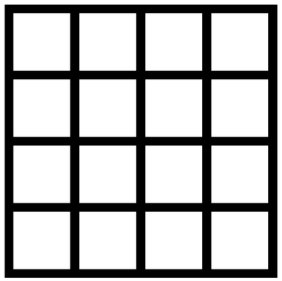
Really ?



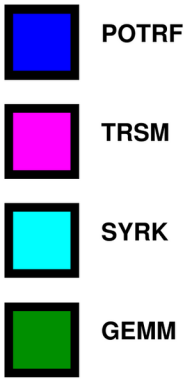
Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

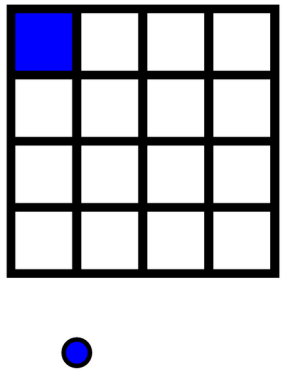


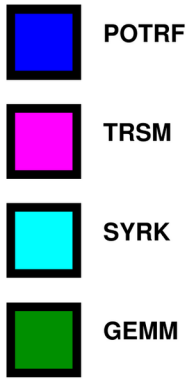
Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



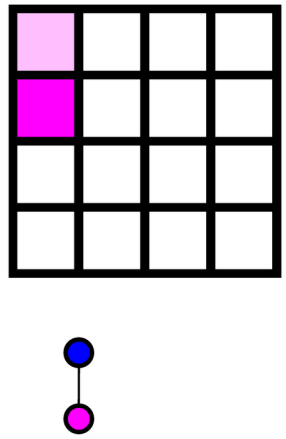


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

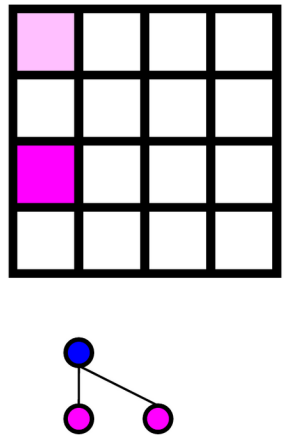


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

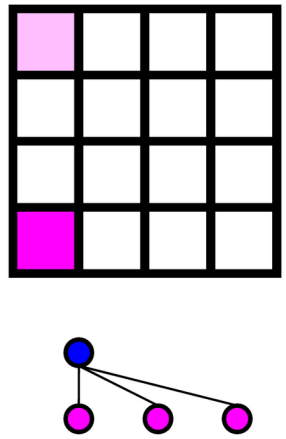


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

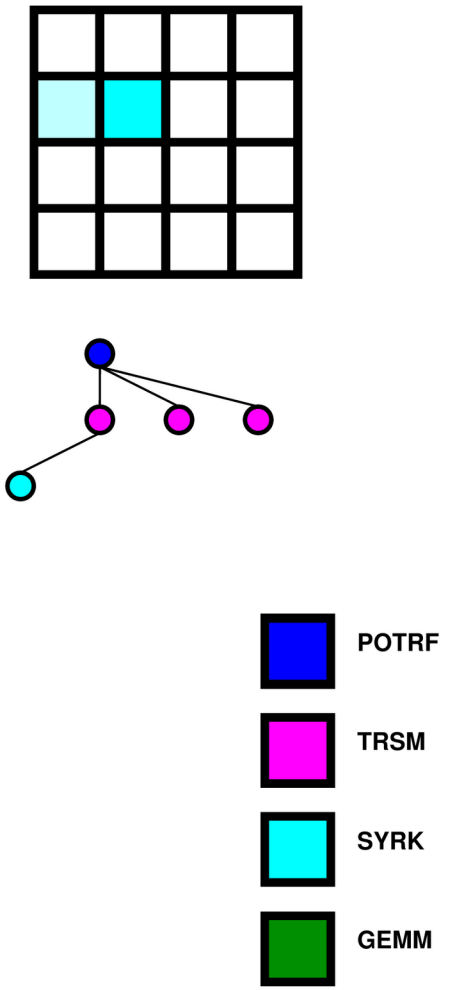


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

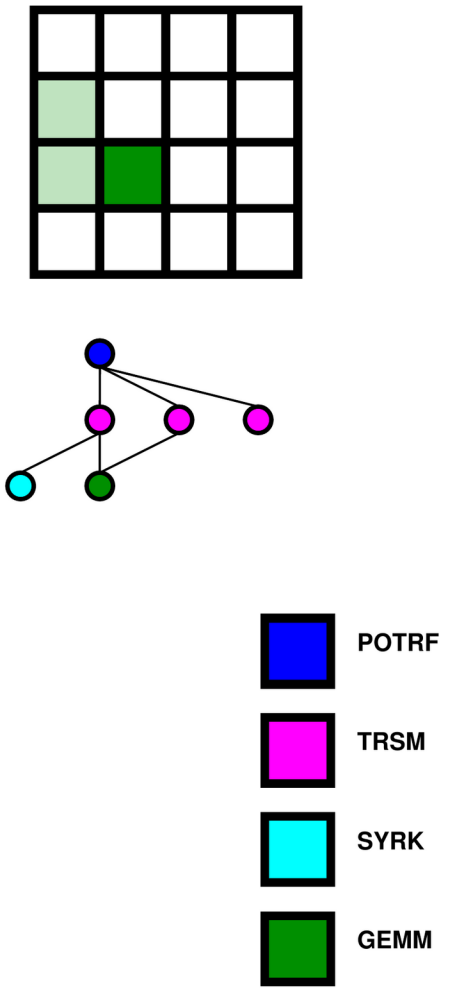


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

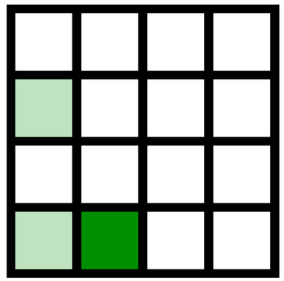


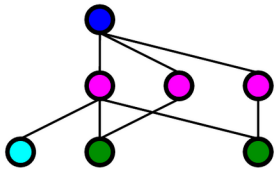
Expressing a task graph

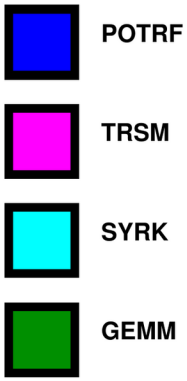
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```





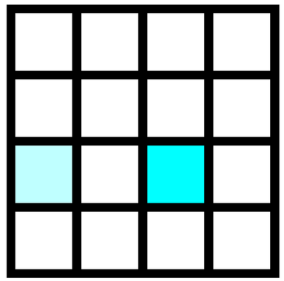


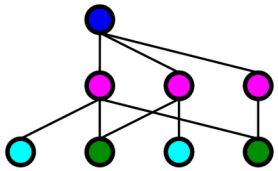
Expressing a task graph

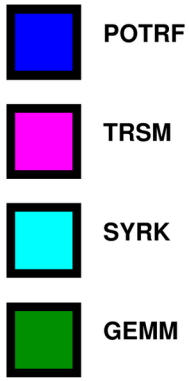
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```





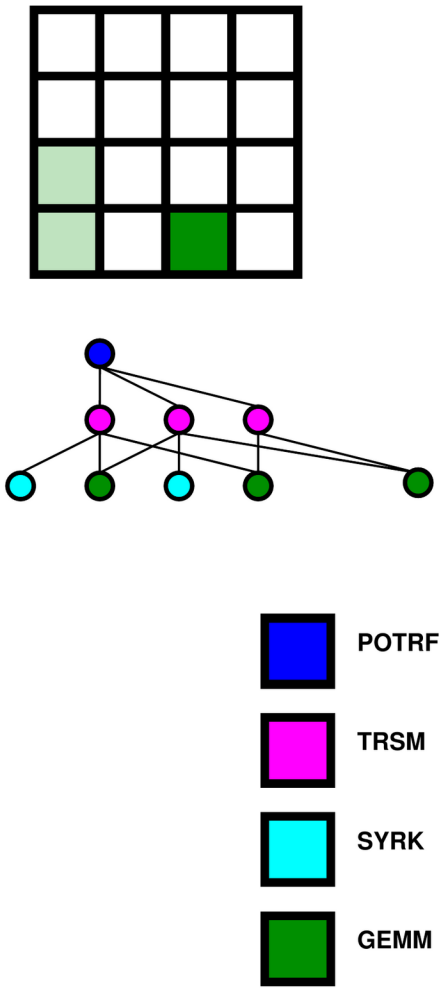


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

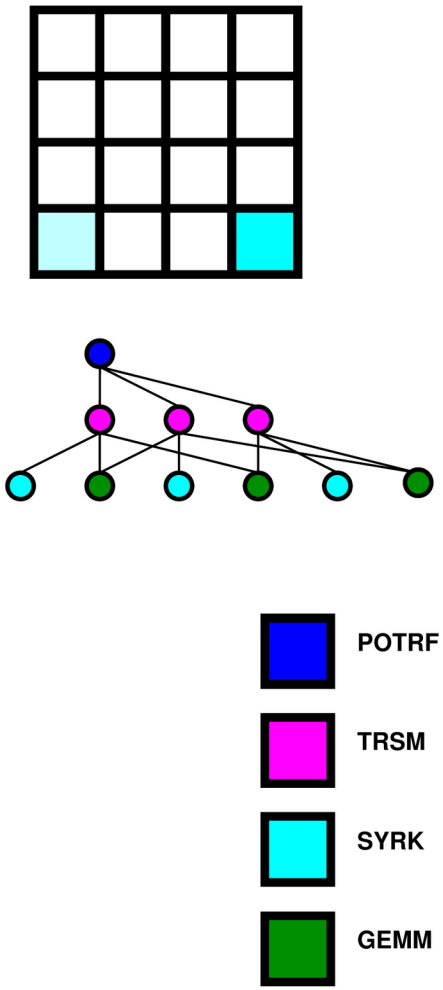


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

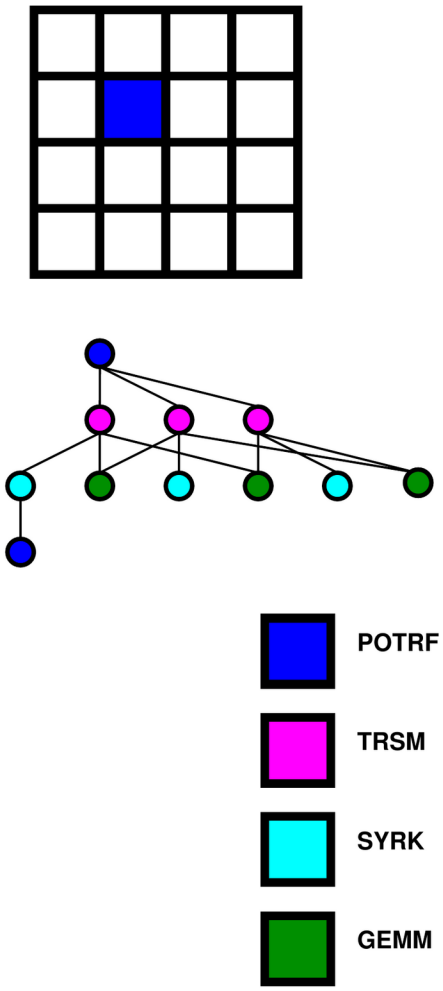


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

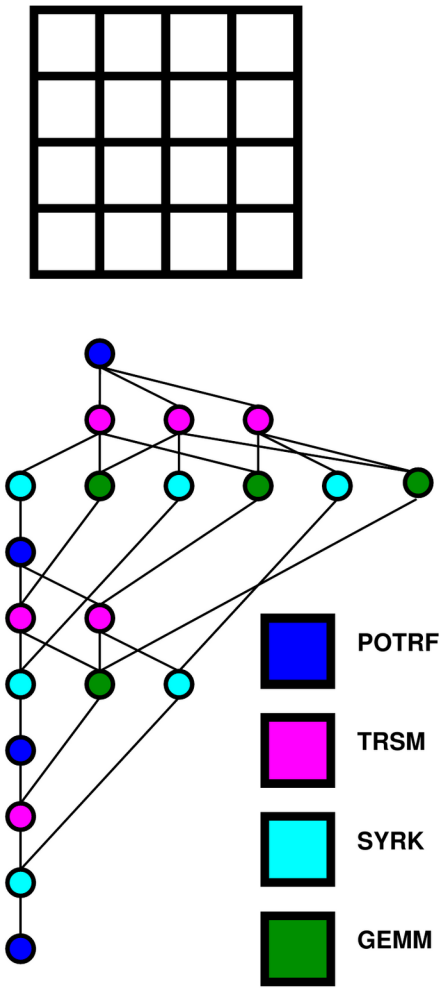


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



Write your application as a task graph

Even if using a sequential-looking source code

➔ Portable performance

Sequential Task Flow (STF)

- Algorithm remains the same on the long term
- Can debug the sequential version.
- Only kernels need to be rewritten
  - BLAS libraries, multi-target compilers
- Runtime will handle parallel execution

Task-based programming

- Needs code restructuring
  - Split computation into tasks
    - BLAS, typically
    - Supposed to have “stable” performance
- Constraining
  - No global variables
    - Mandatory for GPUs
- Actually… functional programming

So a good move, in the end ☺

- Have to accept constraints and losing control

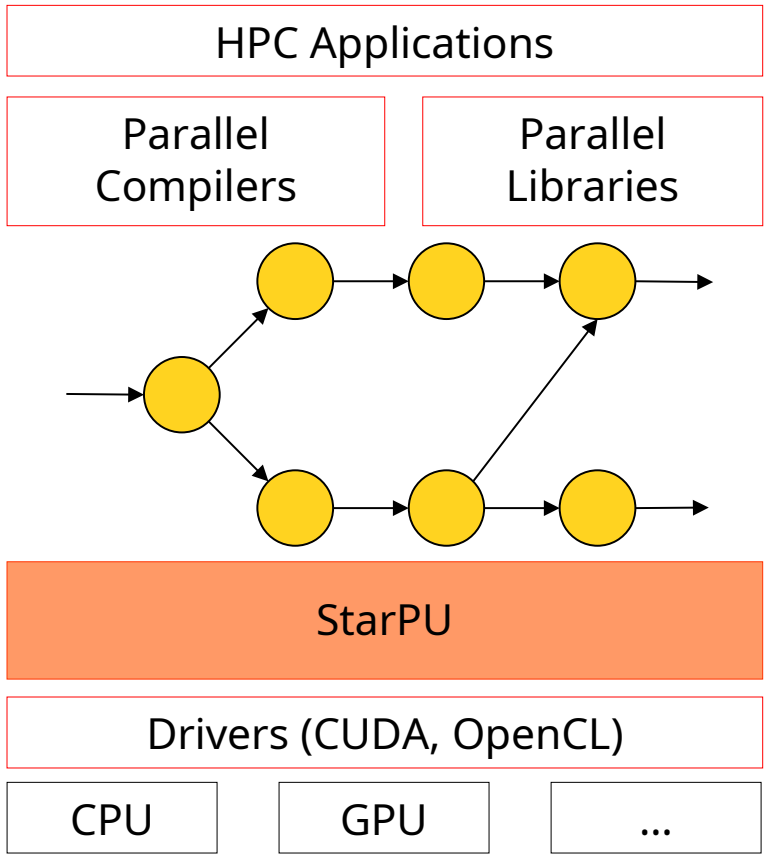
Just like we did when moving from assembly to high-level languages

Overview of StarPU

The StarPU runtime system

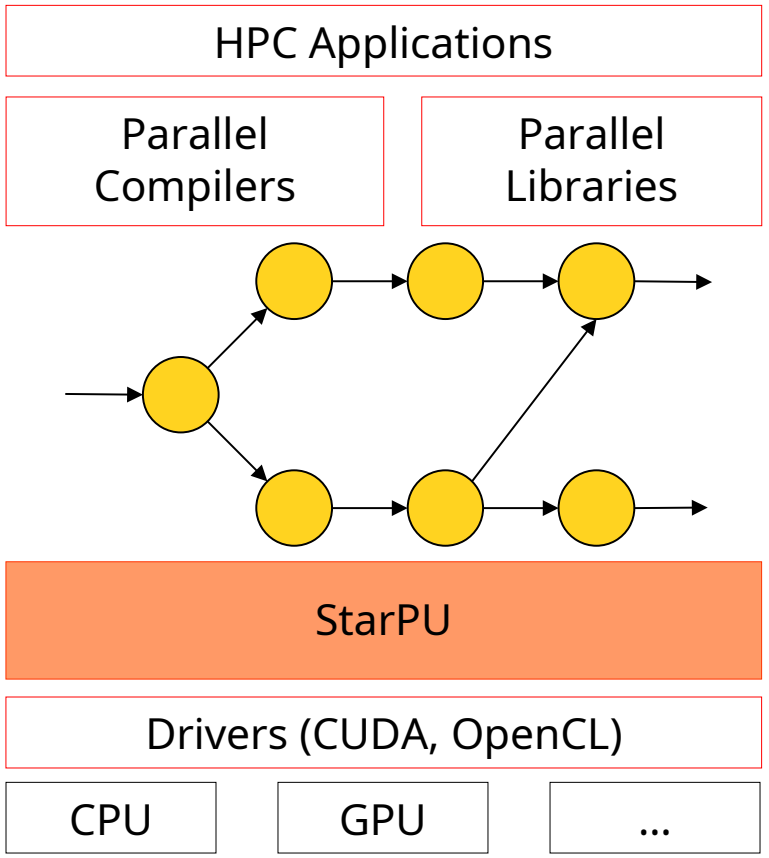
The need for runtime systems

- "do dynamically what can't be done statically anymore"
- Compilers and libraries generate (graphs of) tasks
  - Additional information is welcome!
- StarPU provides
  - Task scheduling
  - Memory management



Data management

- StarPU provides a Virtual Shared Memory (VSM) subsystem (aka DSM)
  - Replication
  - Consistency
  - Single writer
    - Or reduction, ...
- Input & ouput of tasks = reference to VSM data

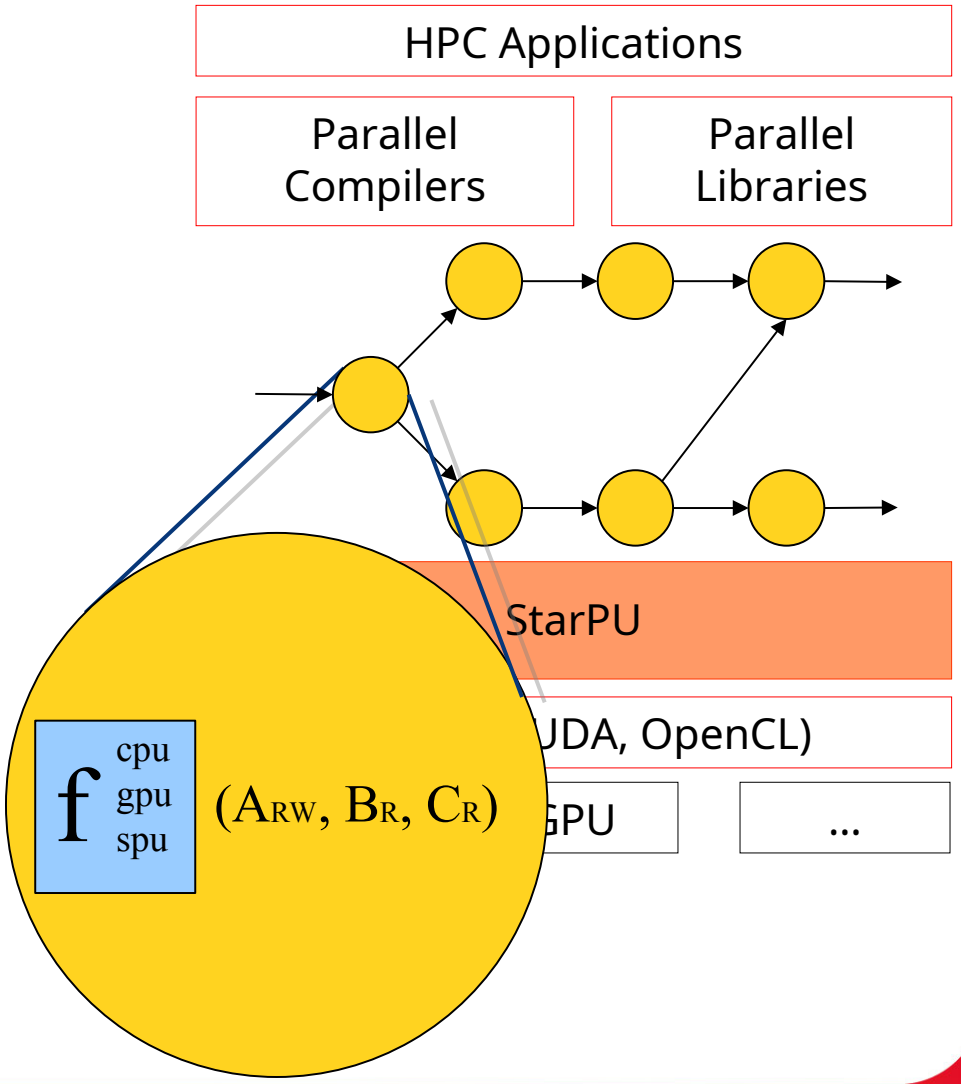


The StarPU runtime system

Task scheduling

- Tasks =
  - Data input & output
    - Reference to VSM data
  - Multiple implementations
    - E.g. CUDA + CPU implementation
  - Non-preemptible
  - Dependencies with other tasks

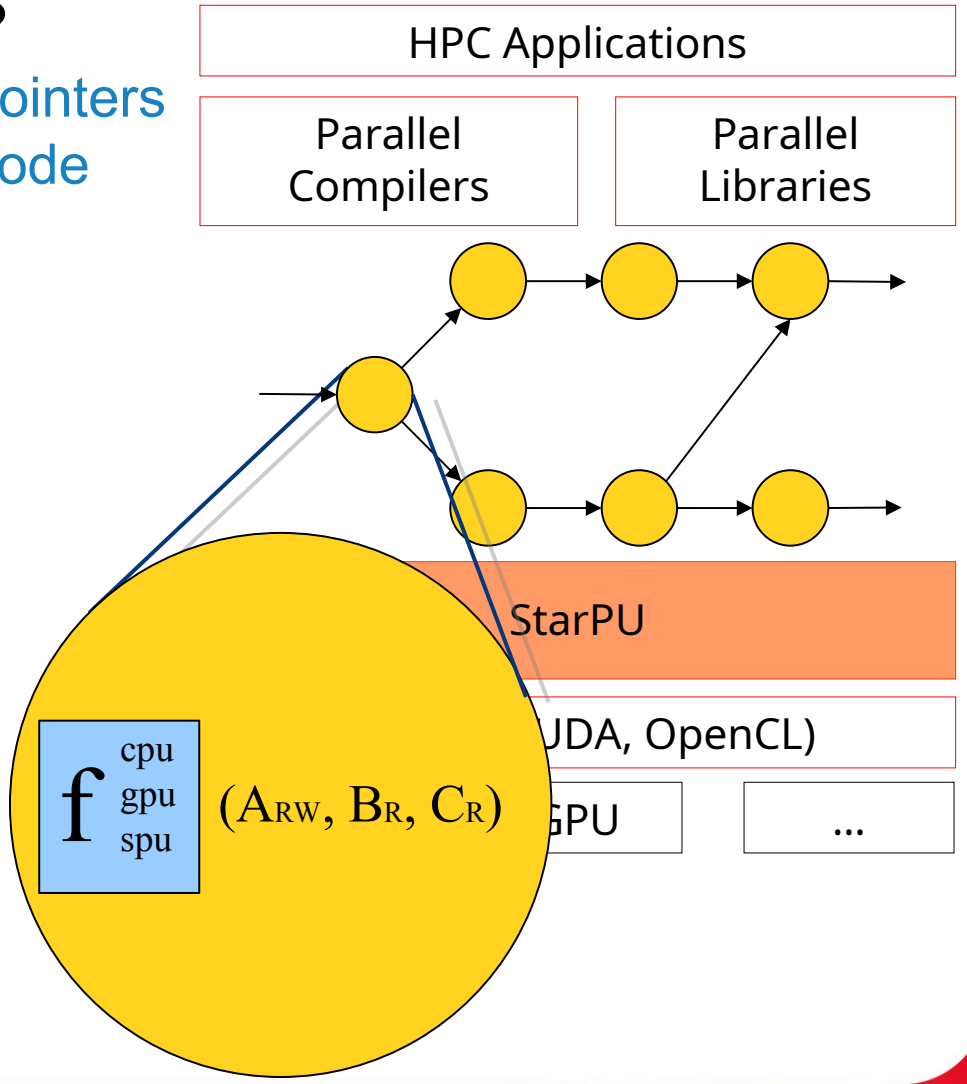
- StarPU provides an Open Scheduling platform
  - Scheduling algorithm = plug-ins



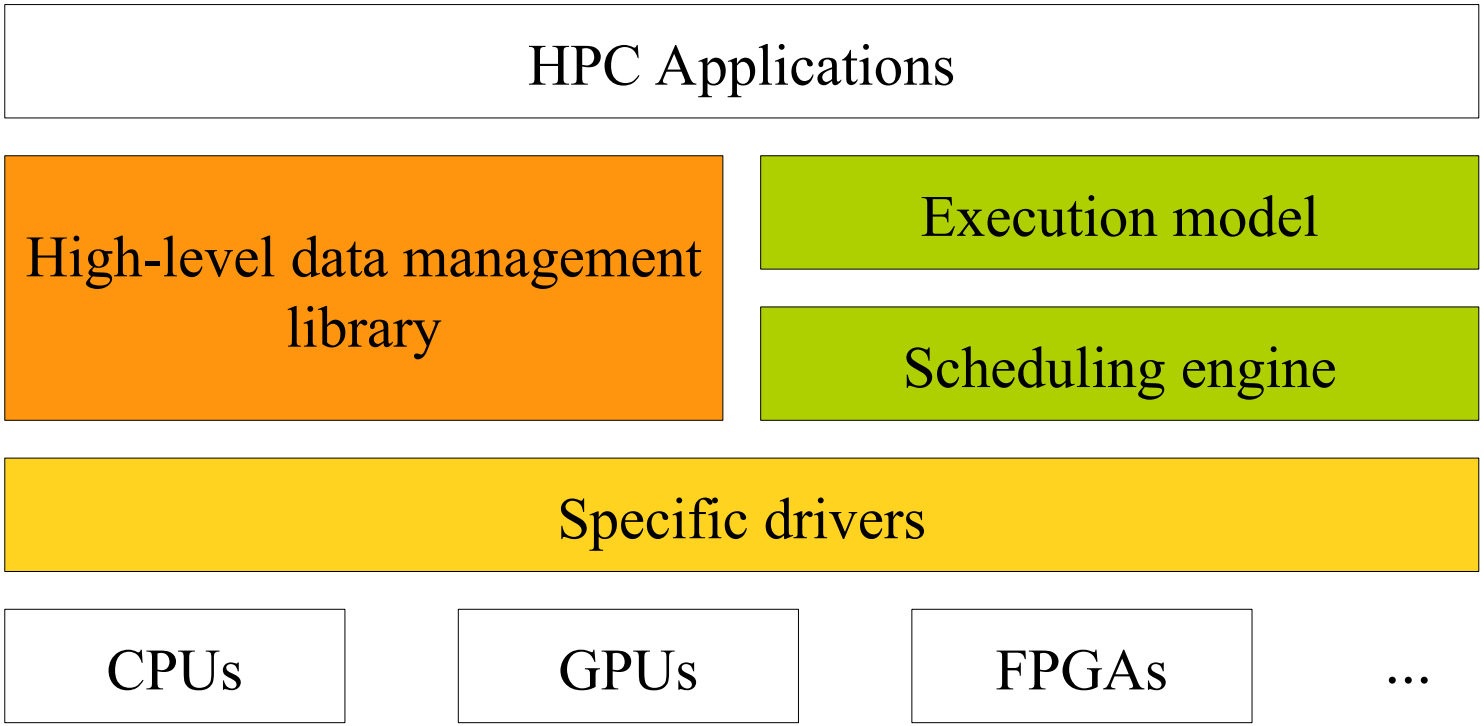
The StarPU runtime system

Task scheduling

- Who generates the code ?
  - StarPU Task ~= function pointers
  - StarPU doesn't generate code
- Libraries era
  - PLASMA + MAGMA
  - FFTW + CUFFT…
  - Variants management
- Rely on compilers



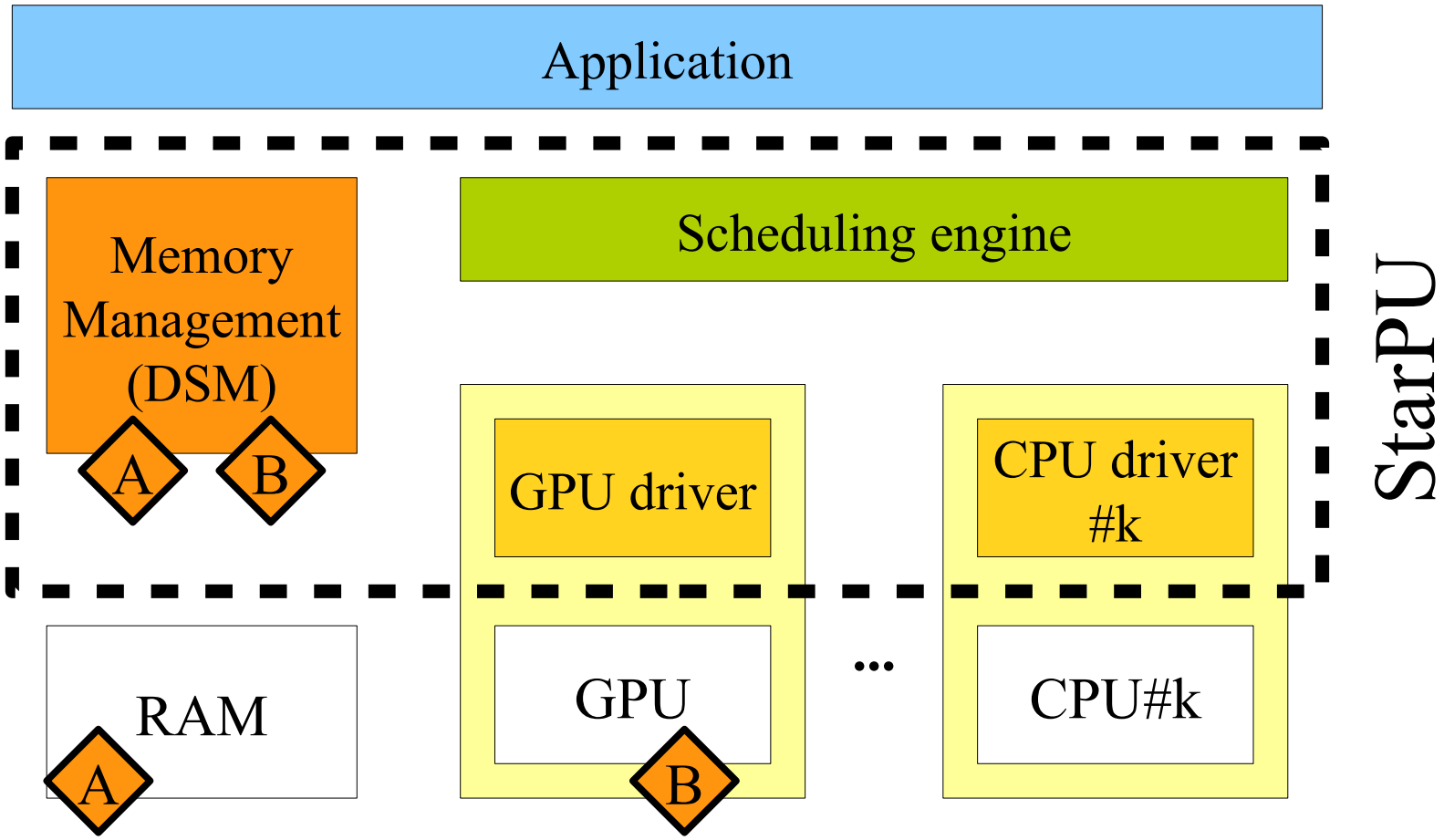
The StarPU runtime system



Mastering CPUs, GPUs, FPGAs … ***PUs → StarPU**

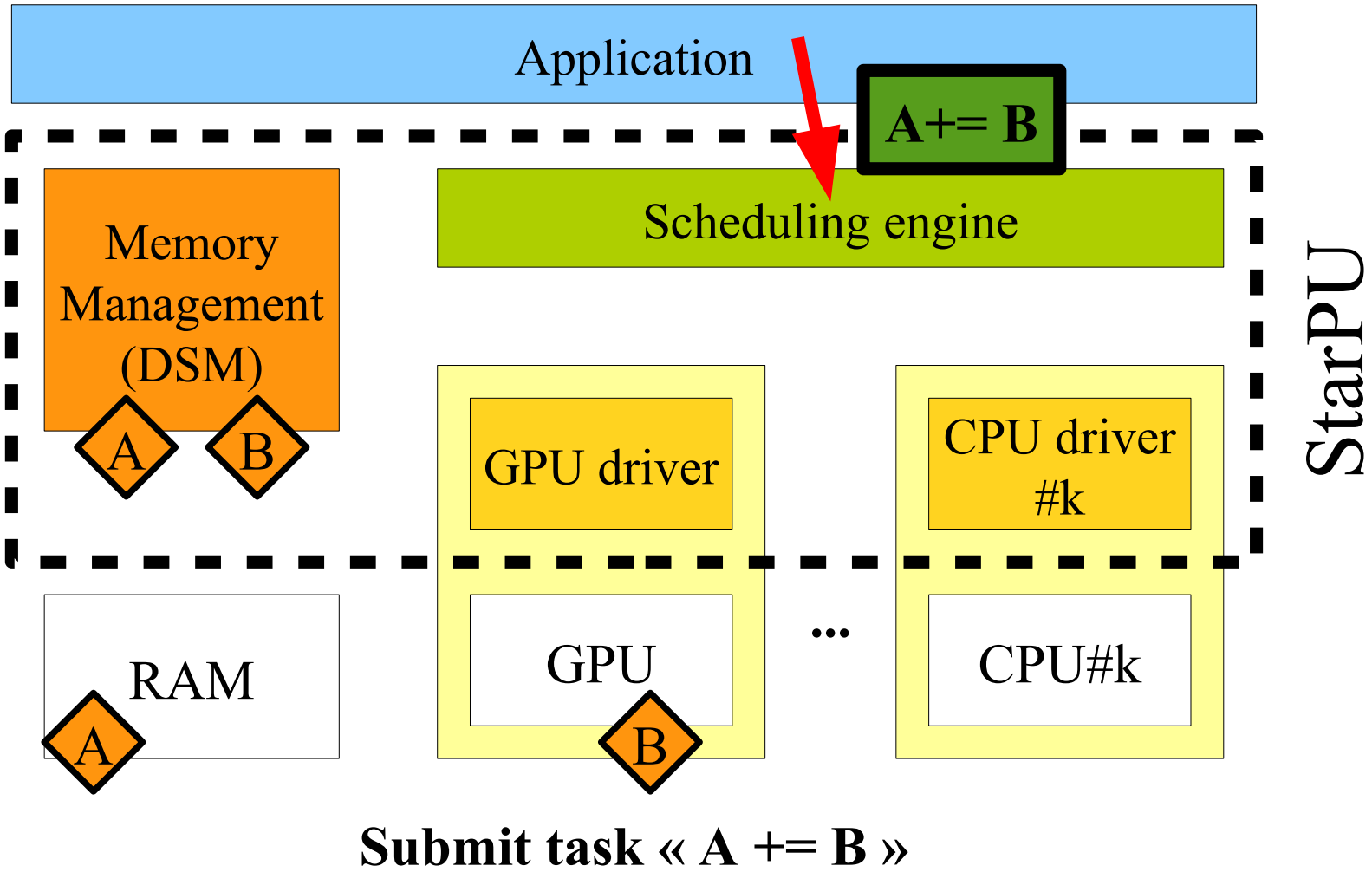
The StarPU runtime system

Execution model



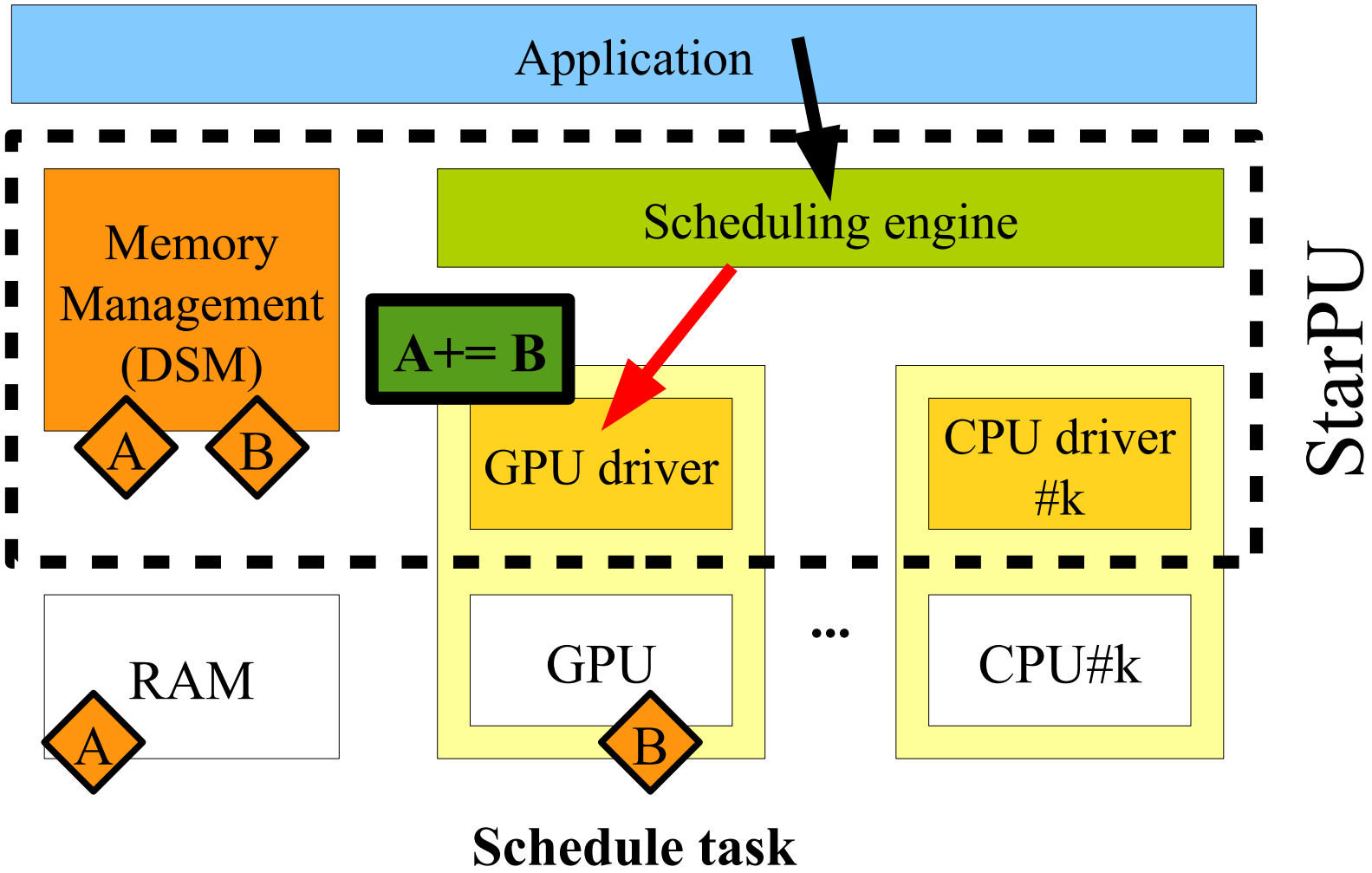
The StarPU runtime system

Execution model



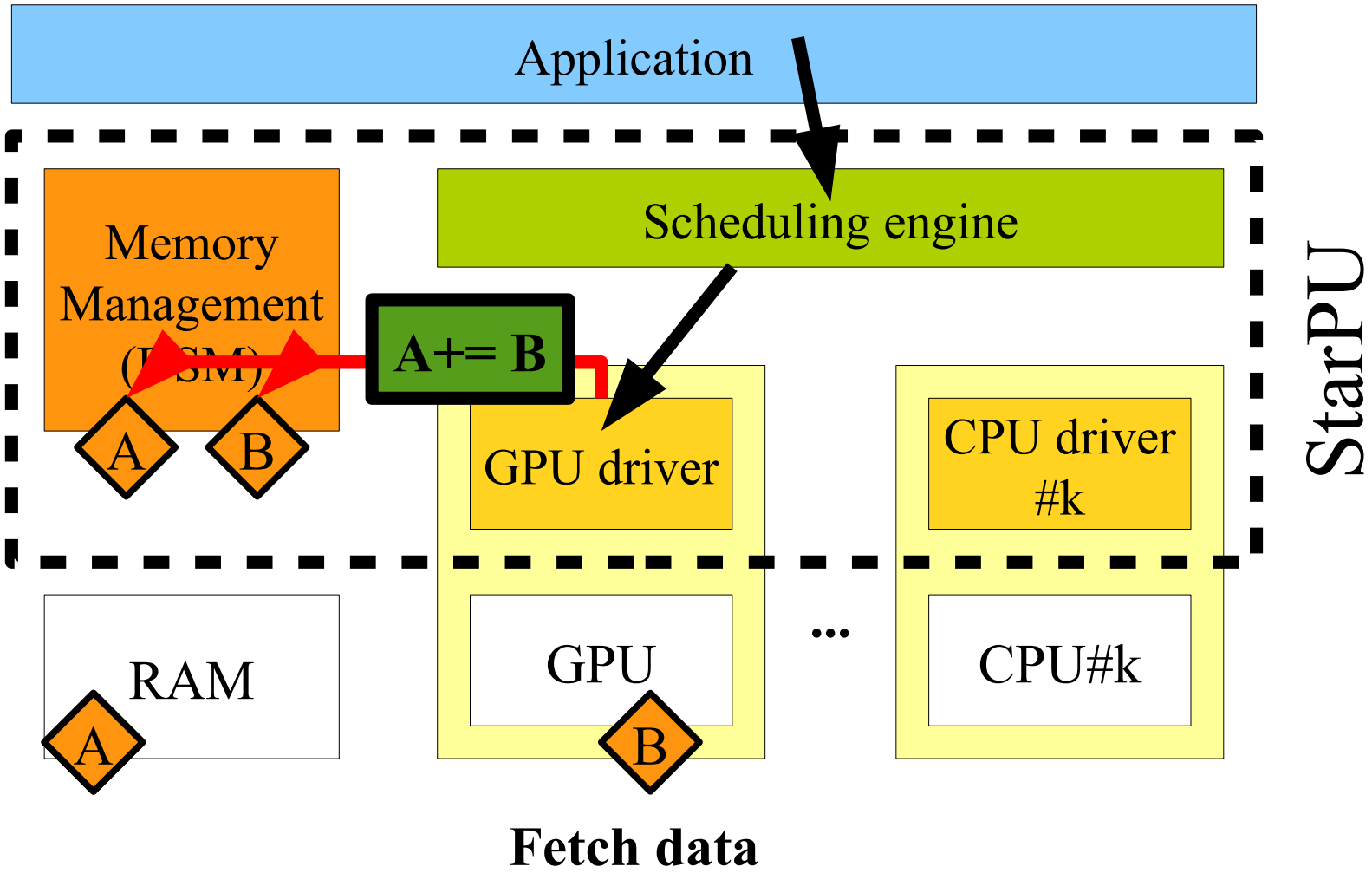
The StarPU runtime system

Execution model



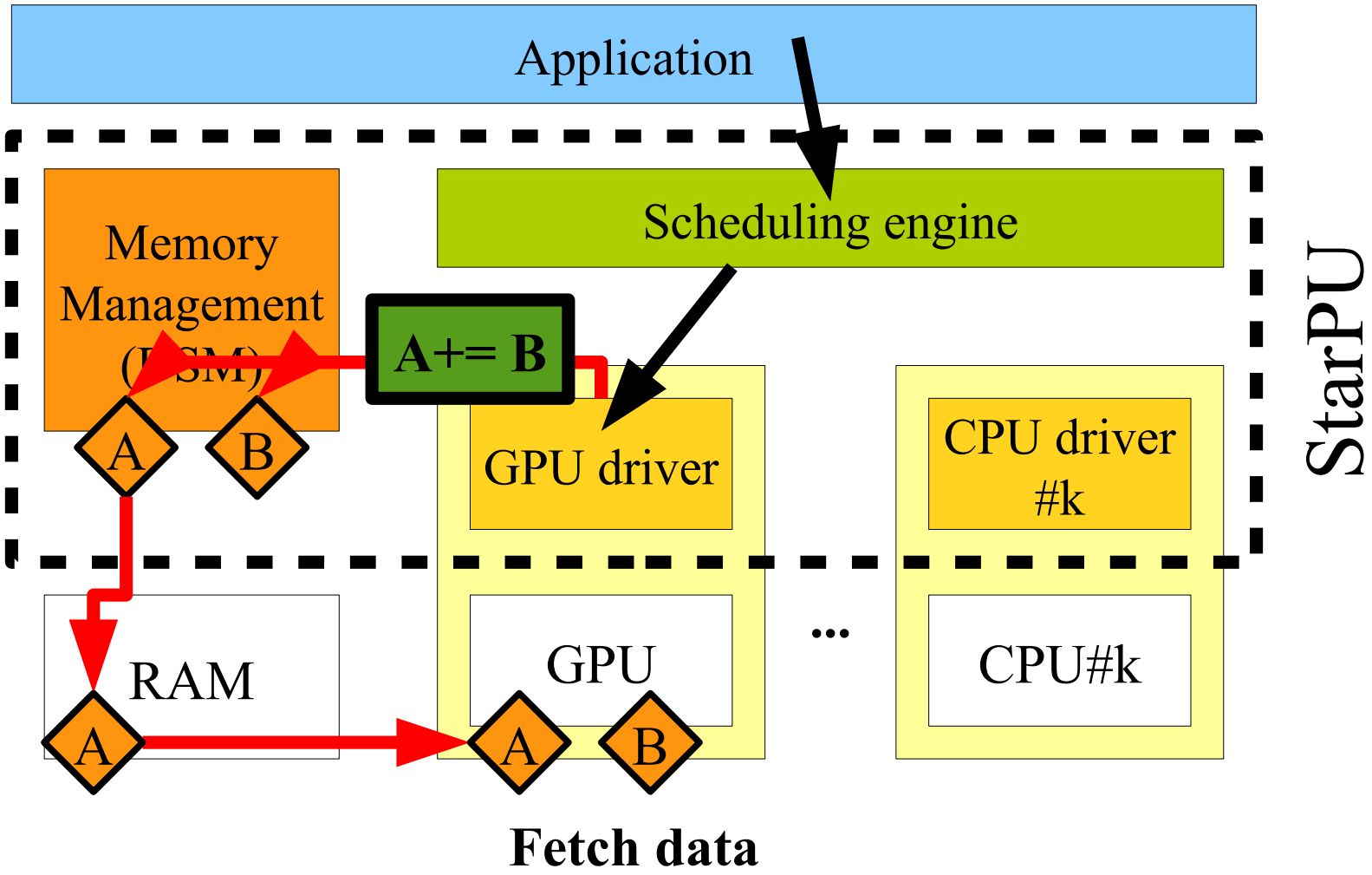
The StarPU runtime system

Execution model



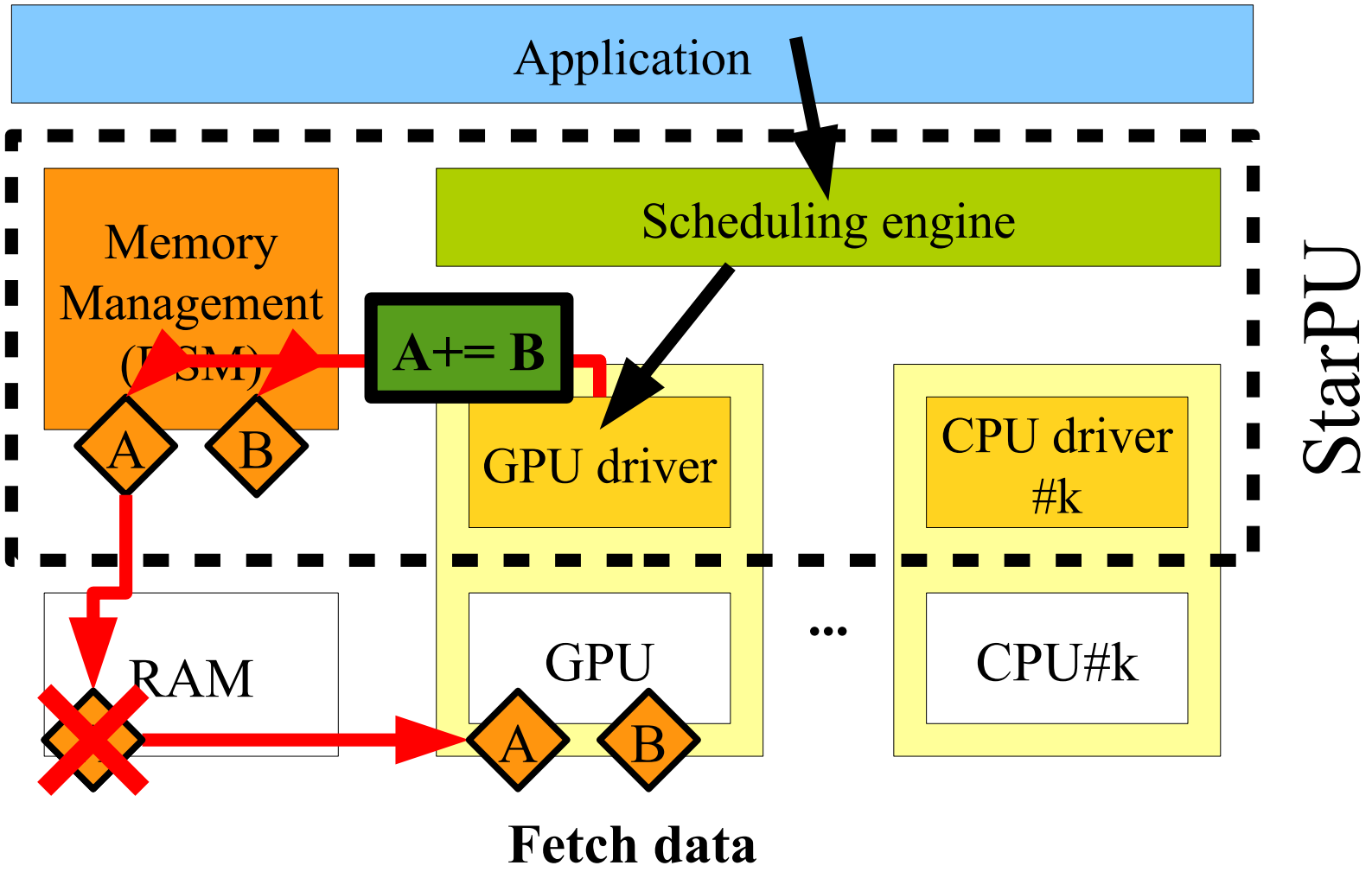
The StarPU runtime system

Execution model



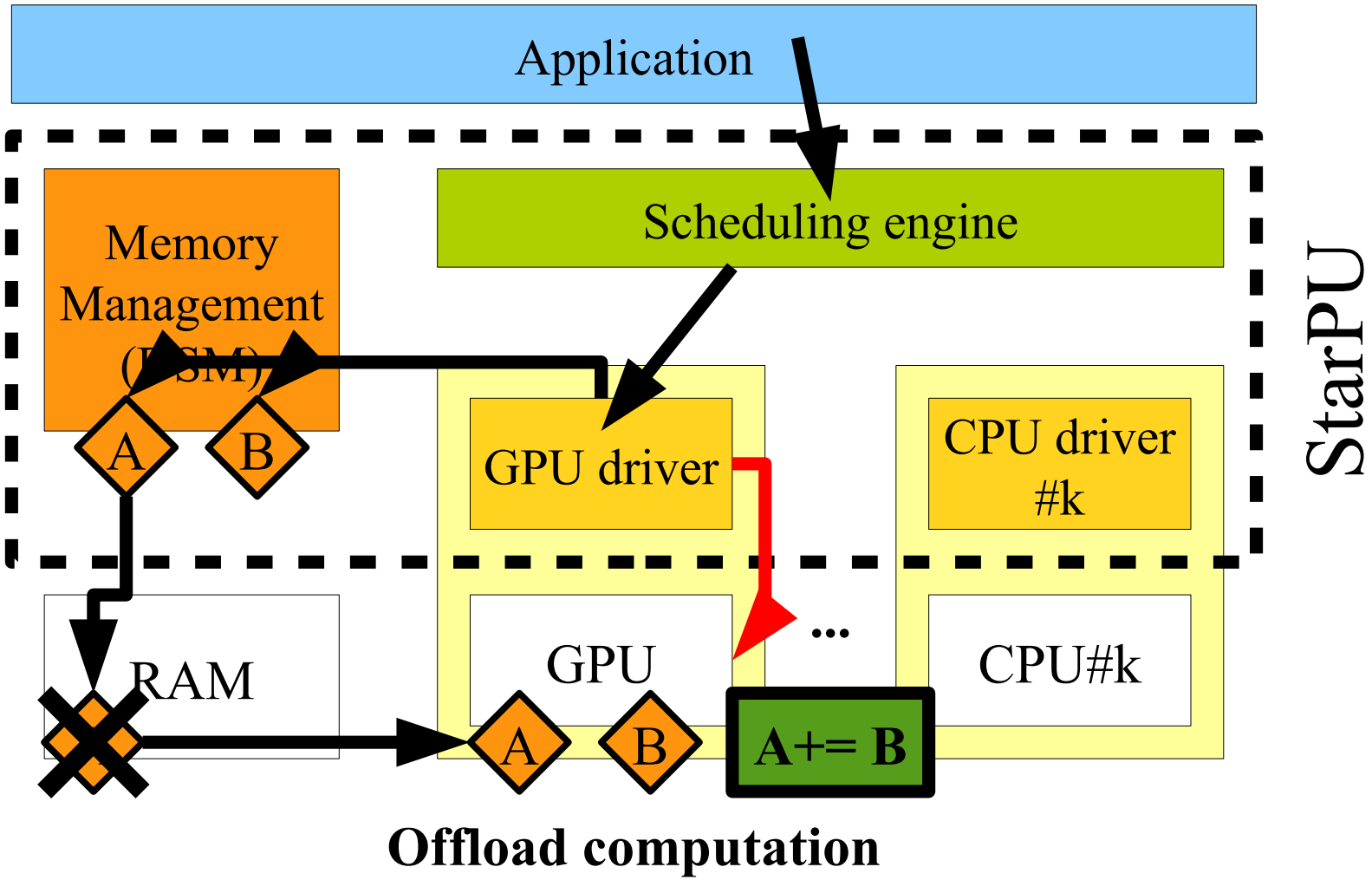
The StarPU runtime system

Execution model



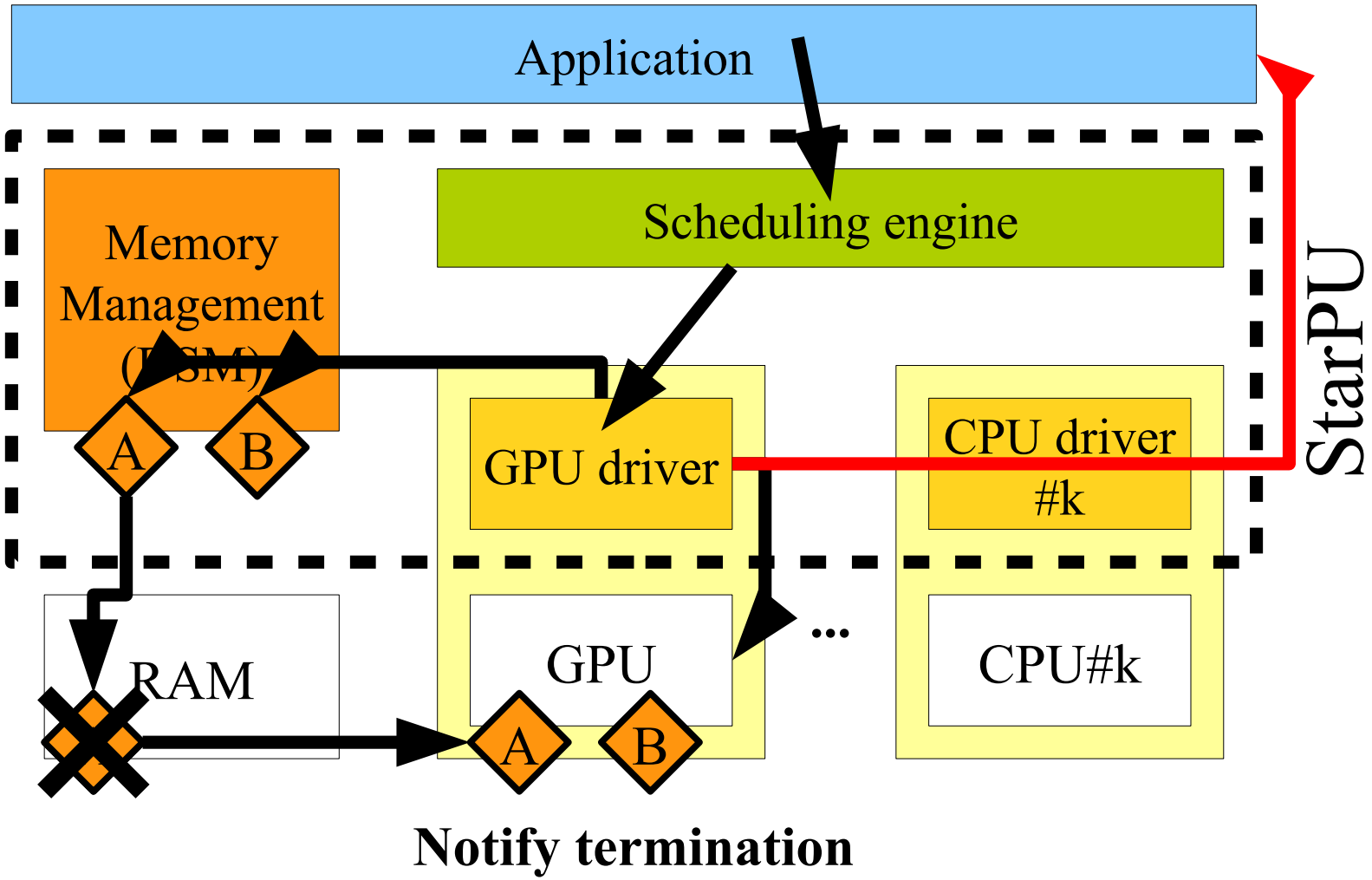
The StarPU runtime system

Execution model



The StarPU runtime system

Execution model



The StarPU runtime system

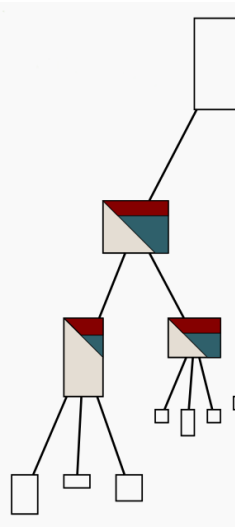
Development context

- History
  - Started in 2008
    - PhD Thesis of Cédric Augonnet
  - StarPU main core ≈ 70k lines of code
  - Written in C
- Open Source
  - Released under LGPL
  - Sources freely available
    - git repository and nightly tarballs
    - See https://starpu.gitlabpages.inria.fr/
  - Open to external contributors
- [HPPC'08]
- [Europar'09] – [CCPE'11],... >1900 citations

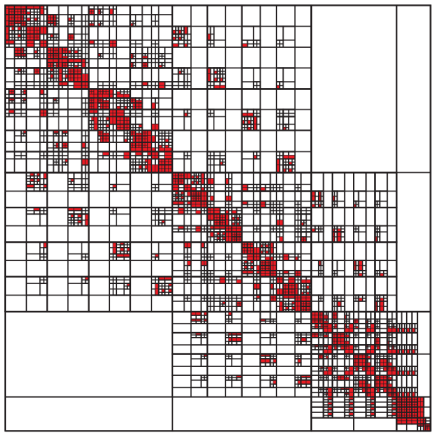
The StarPU runtime system

Success stories

Task-based programming actually makes things easier!



- QR-Mumps (sparse linear algebra)
  - Non-task version: only 1D decomposition
  - Task version: 2D decomposition, flurry of parallelism
    - With seamless memory control
- H-Matrices (compressed linear algebra, AirBus)
  - Out-of-core support
    - Could run cases unachievable before
    - e.g. 1600 GB matrix with 256 GB memory
  - Shipped to AirBus customers
- Implemented CFD, FMM, CG, stencils, …



The StarPU runtime system

Supported platforms

- Supported architectures
  - Multicore CPUs (x86, PPC, ...)
  - NVIDIA GPUs
  - OpenCL devices (eg. AMD cards)
  - HIP
  - FPGA (ongoing)
  - Old Intel Xeon Phi (MIC), SCC, Kalray MPPA, Cell (decommissioned)
- Supported Operating Systems
  - Linux
  - Mac OS
  - Windows

Scaling a vector

Data registration

- Register a piece of data to StarPU (vector, matrix, ...)

```
float array[NX];
for (unsigned i = 0; i < NX; i++)
    array[i] = 1.0f;

starpu_data_handle vector_handle;
starpu_vector_data_register(&vector_handle, 0,
        array, NX, sizeof(vector[0]));
```

- Submit tasks….
- Unregister data
  - starpu_data_unregister(**vector_handle**);

Scaling a vector

Defining a codelet

- Codelet = multi-versionned kernel
  - Function pointers to the different kernels
  - Number of data parameters managed by StarPU
  - Access modes

```
starpu_codelet scal_cl = {
    .cpu_funcs = { scal_cpu_func, scal_cpu_func_sse },
    .cuda_funcs = { scal_cuda_func },
    .opencl_funcs = { scal_opencl_func },
    .nbuffers = 1,
    .modes = { STARPU_RW },
};
```

Scaling a vector

Defining a codelet (2)

- CPU kernel

```
void scal_cpu_func(void *buffers[], void *cl_arg)
{
    struct starpu_vector_interface_s *vector = buffers[0];

    unsigned n = STARPU_VECTOR_GET_NX(vector);
    float *val   = STARPU_VECTOR_GET_PTR(vector);

    float *factor = cl_arg;

    for (int i = 0; i < n; i++)
        val[i] *= *factor;
}
```

Scaling a vector

Defining a task, starpu_insert_task helper

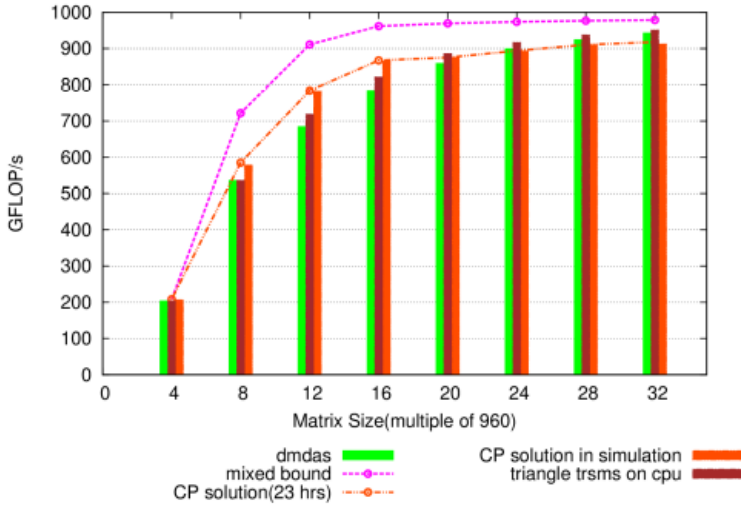
- Define a task that scales the vector by a constant

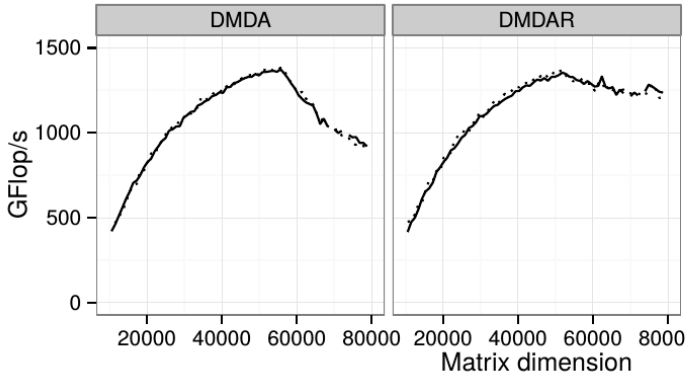
```
float factor = 3.14;

starpu_insert_task(
        &scal_cl,
        STARPU_RW, vector_handle,
        STARPU_VALUE,&factor,sizeof(factor),
        0);
```

Task-based support

Then all of this comes “for free” :

- Task/data scheduling
  - Pipelining
  - Load balancing
  - GPU memory limitation management
  - Data prefetching
- Performance bounds
- Distributed execution through MPI
- High-level performance analysis
- Out-of-core : optimized swapping to disk
- Debugging sequential execution
- Reproducible performance simulation





Applications on top of StarPU

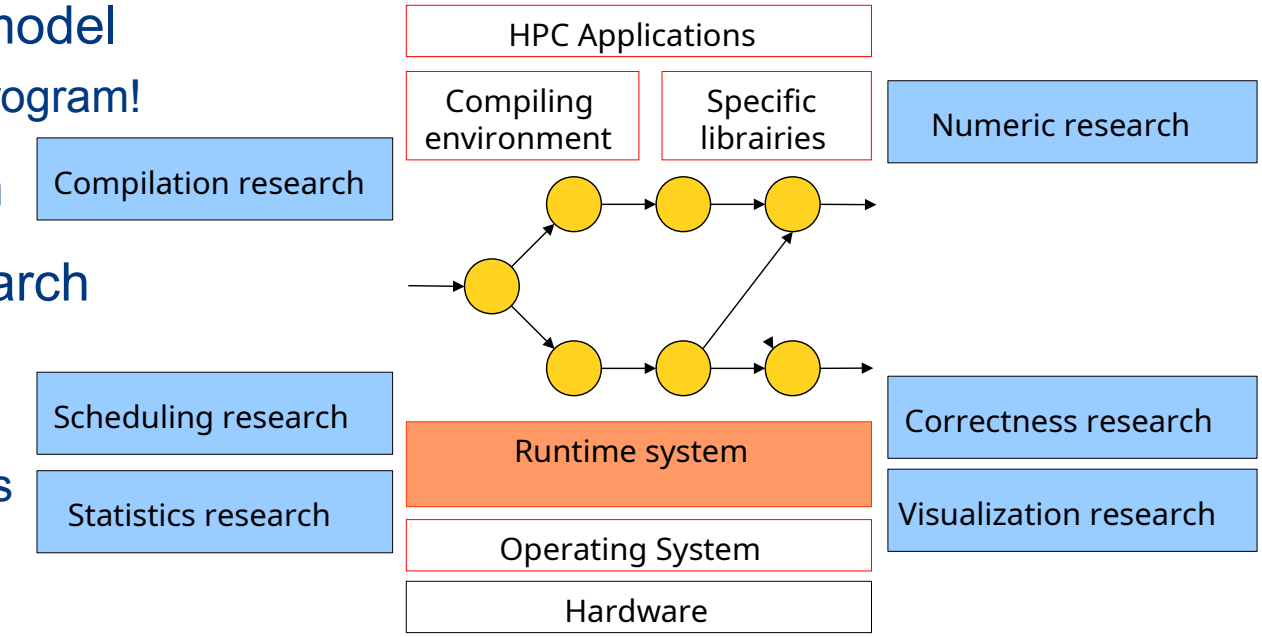
Using CPUs, GPUs, distributed, out of core, ...

- Dense linear algebra
  - Cholesky, QR, LU, ... : Chameleon (based on Plasma/Magma)
- Sparse linear algebra
  - QR_MUMPS
  - PaStiX
- Compressed linear algebra
  - BLR, h-matrices
- Fast Multipole Method
  - ScalFMM
- Conjugate Gradient
- Other programming models : Data flow, skeletons
  - SignalPU, SkePU
- ...

Conclusion

Task graphs

- Nice programming model
  - Keep sequential program!
- Optimized execution
- Playground for research
  - Runtime
  - Scheduling
  - Numeric algorithms
  - Statistics
  - Correctness
- Used for various real-world computations
  - Cholesky/QR/LU (dense/sparse/compressed), stencil, CG, CFD, FMM…



http://starpu.gitlabpages.inria.fr/tutorials/