

# Extending the StarSs Programming Model to Platforms with Multiple Hardware Accelerators

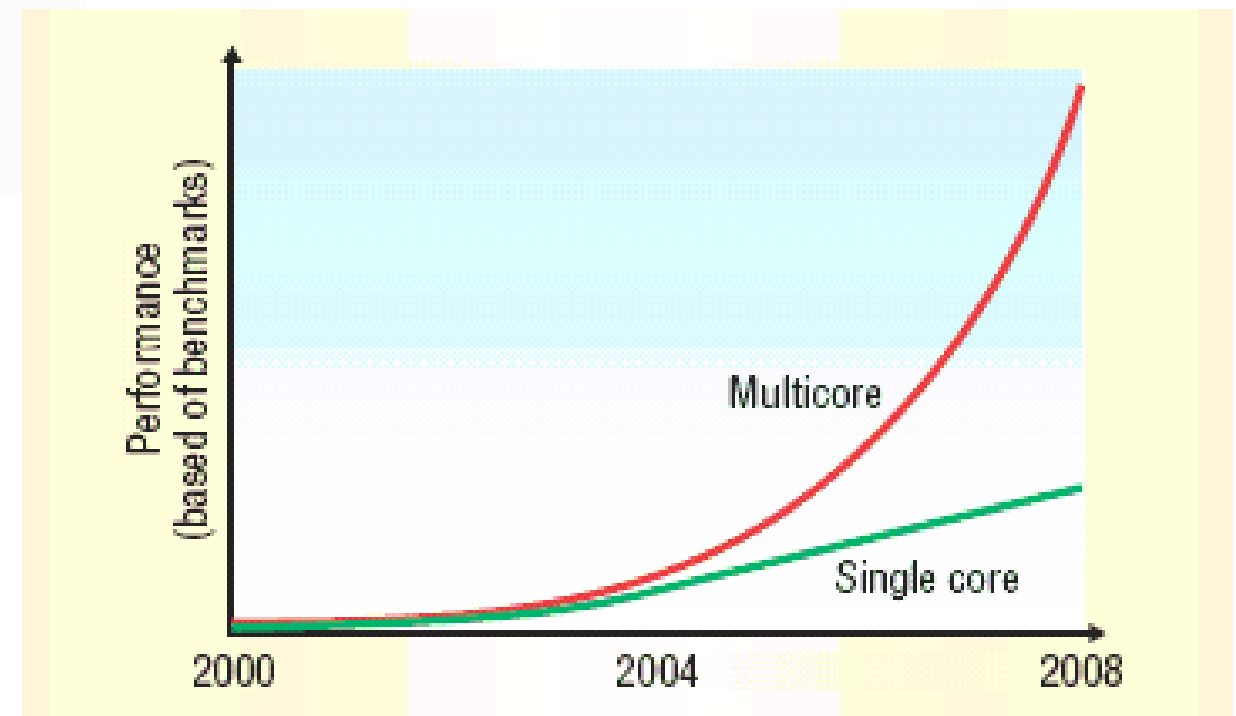
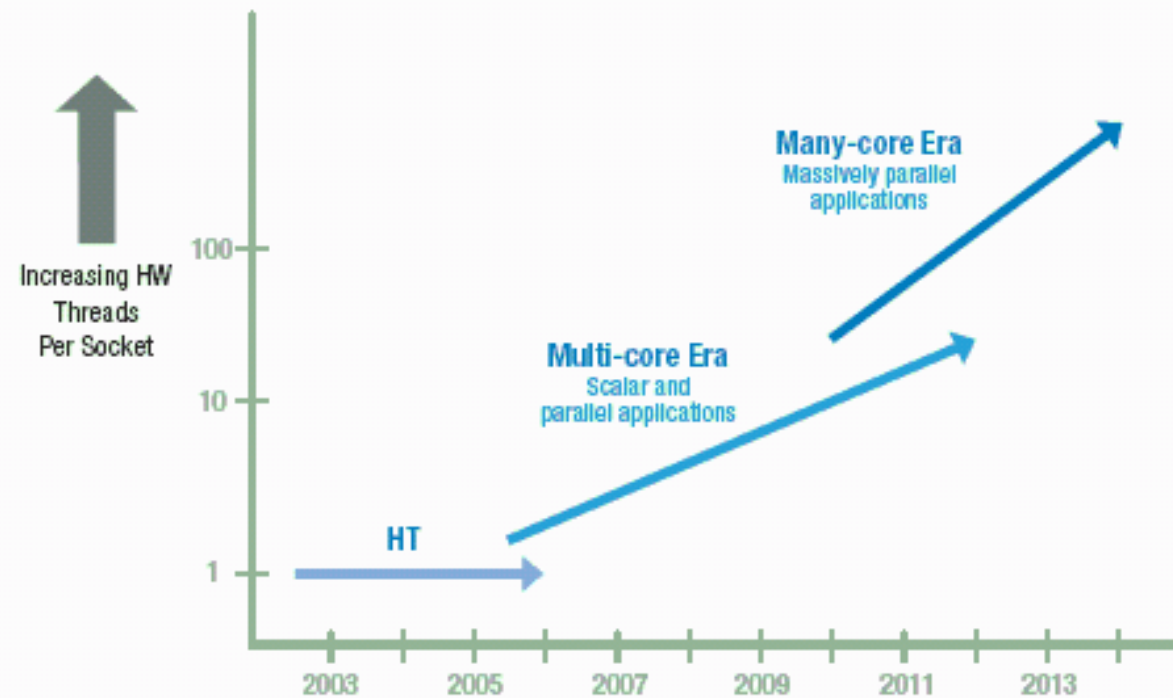
**Barcelona Supercomputing  
Center**



**Universidad Jaime I de  
Castellón**

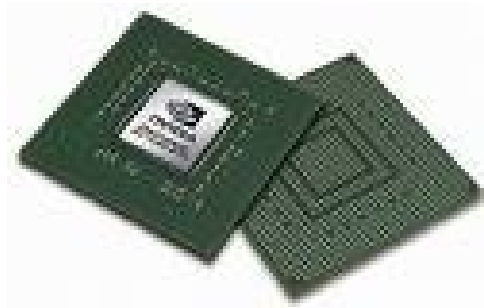


# Motivation



\* Source: Platform 2015: Intel® Processor and Platform Evolution for the Next Decade, Intel White Paper

- A few many-core architectures are already available:



- A coarser-grain parallelism is present in platforms with multiple accelerators:



- Distributed memory
- No hardware coherence mechanism
- Heterogeneity

- Program these architectures as message-passing?

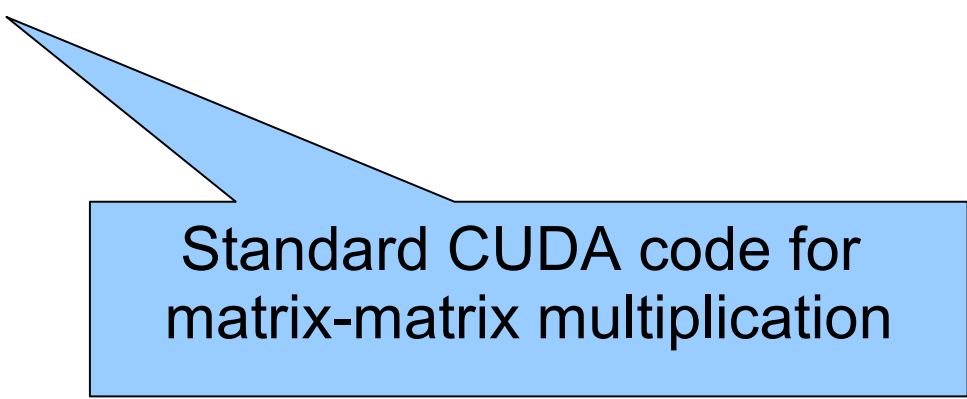


- Difficult!
  - Rewrite existing libraries
  - Likely not the most efficient solution
- Extend the StarSs programming model to these platforms!
    - “Sequential” program (i.e., single accelerator)
    - StarSs run-time extracts coarse-grain parallelism to exploit multiple accelerators

- StarSs tailored to **multi-GPU platforms**
- Similarities between Cell B.E. and multi-GPU platforms:
  - Cell B.E.: PPU + (8) SPU with Local Store
  - Tesla: CPU + (4) GPUs with Global Memory
  - Interconnection buses: EIB vs. PCI Express
- Many techniques from CellSs and SMPsSs also valid for GPUSs
- Minimal additions to the StarSs syntax
- Possibility of hybrid GPU-CPU execution
- Potentially valid for other multi-accelerator platforms or heterogeneous platforms

# GPUSs example: Matrix-matrix product

```
__global__ void matmul( float * A, float * B, float * C ){  
    /* CUDA Code */  
}  
  
int main( void ){  
    /* ... */  
    for (i = 0; i < N; i++)  
        for (j = 0; j < N; j++)  
            for (k = 0; k < N; k++)  
                matmul ( A[i][k], B[k][j], C[i][j] );  
    /* ... */  
}
```



Standard CUDA code for matrix-matrix multiplication

# GPUSs example: Matrix-matrix product

```
#pragma css task input(A[BS][BS], B[BS][BS]) inout(C[BS][BS])
device CUDA(dimGrid, dimBlock)
__global__ void matmul( float * A, float * B, float * C ){

    /* CUDA Code */

}

int main( void ){
    /* ... */
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                matmul ( A[i][k], B[k][j], C[i][j] );
    /* ... */
}
```

## Task definition:

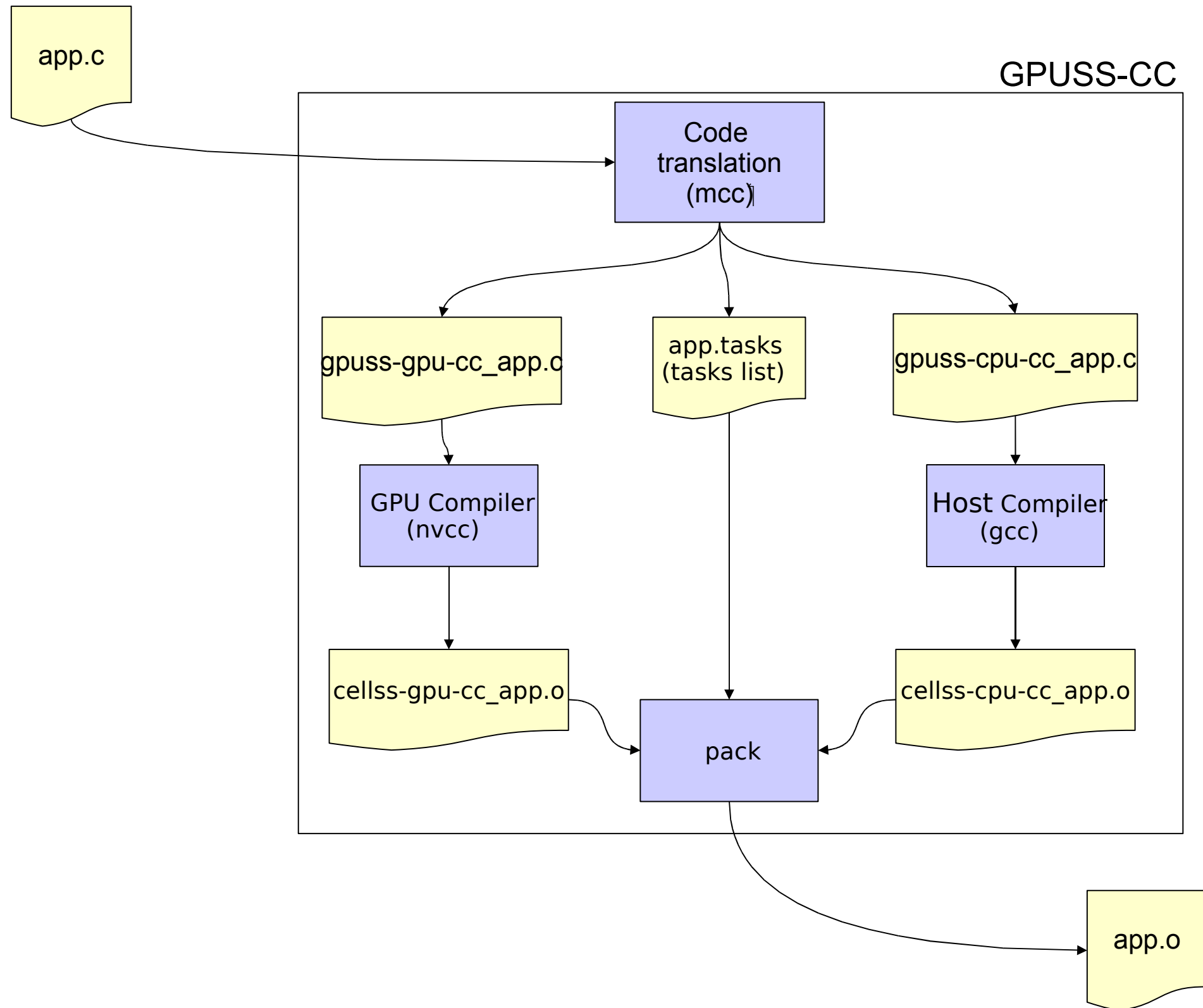
- Input/output parameters
- Type of accelerator
- Execution configuration

## Main program:

- No explicit data transfers or allocation
- No explicit execution configuration
- No changes in the main program

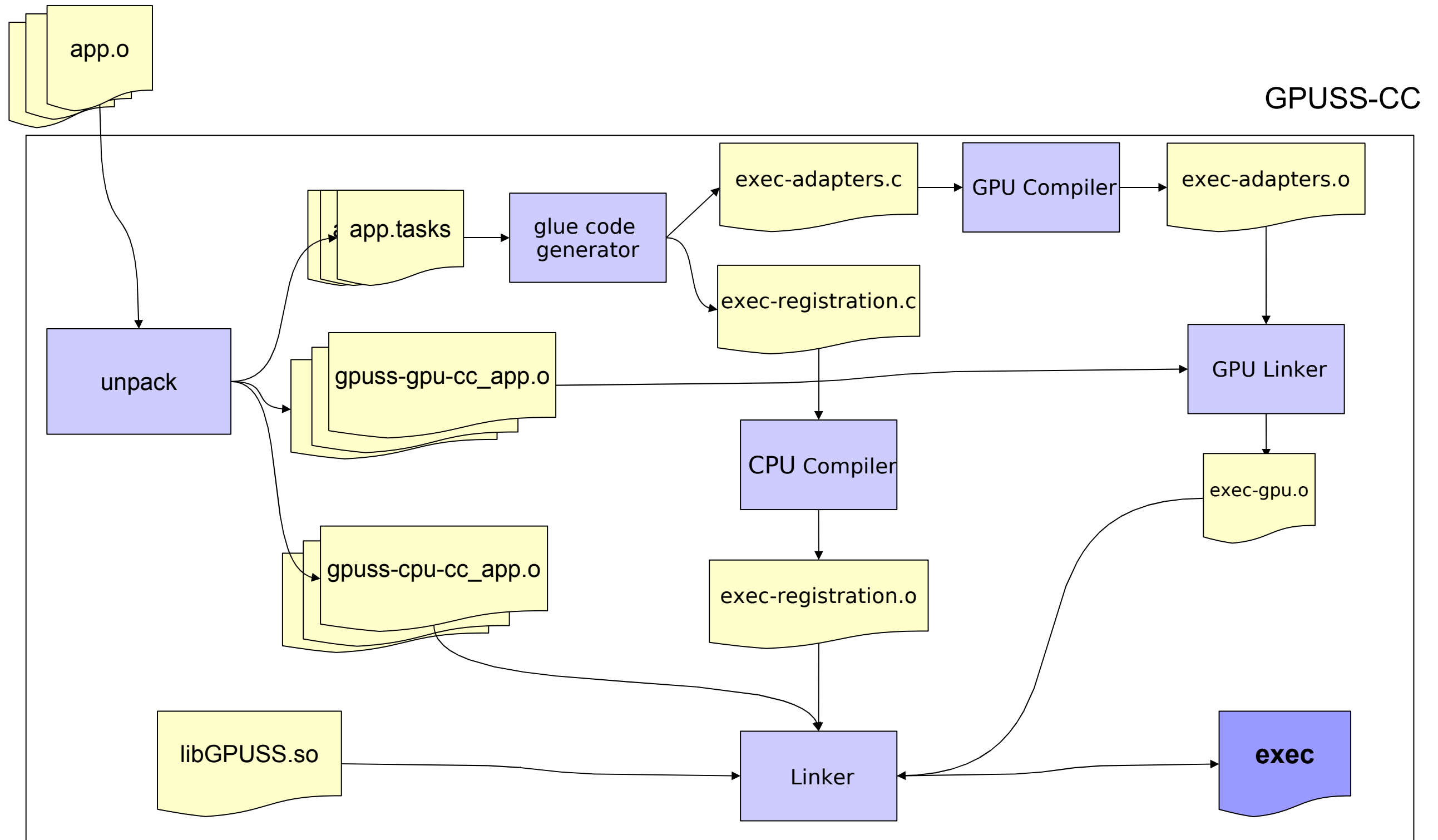


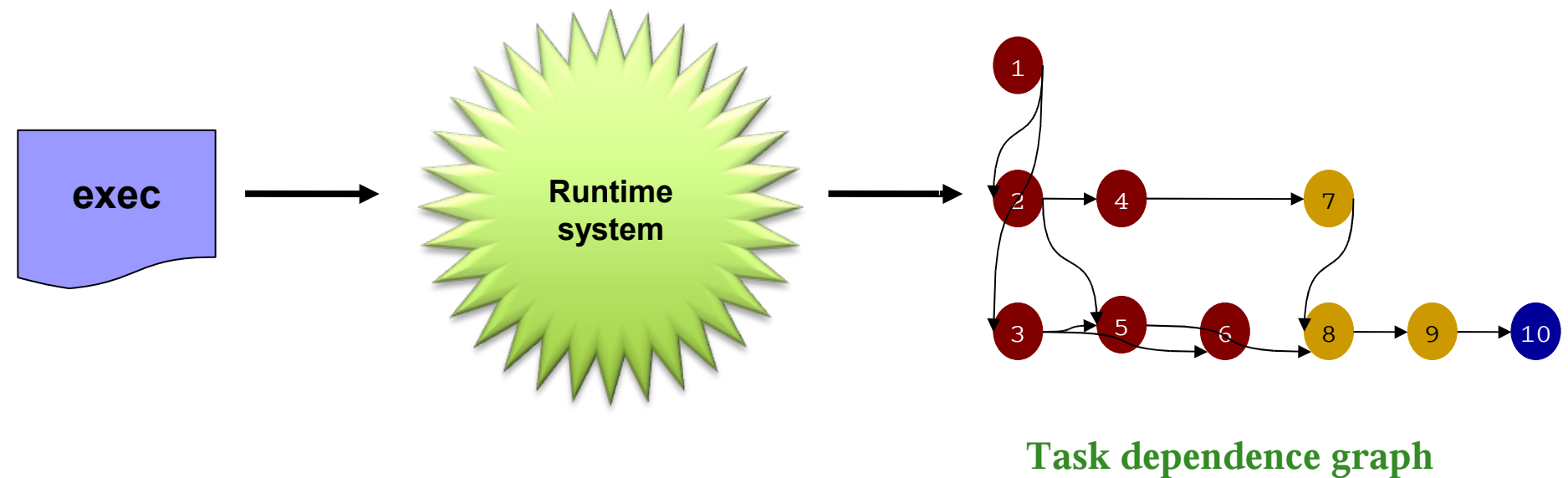
# GPUSs compiler: Compiler phase



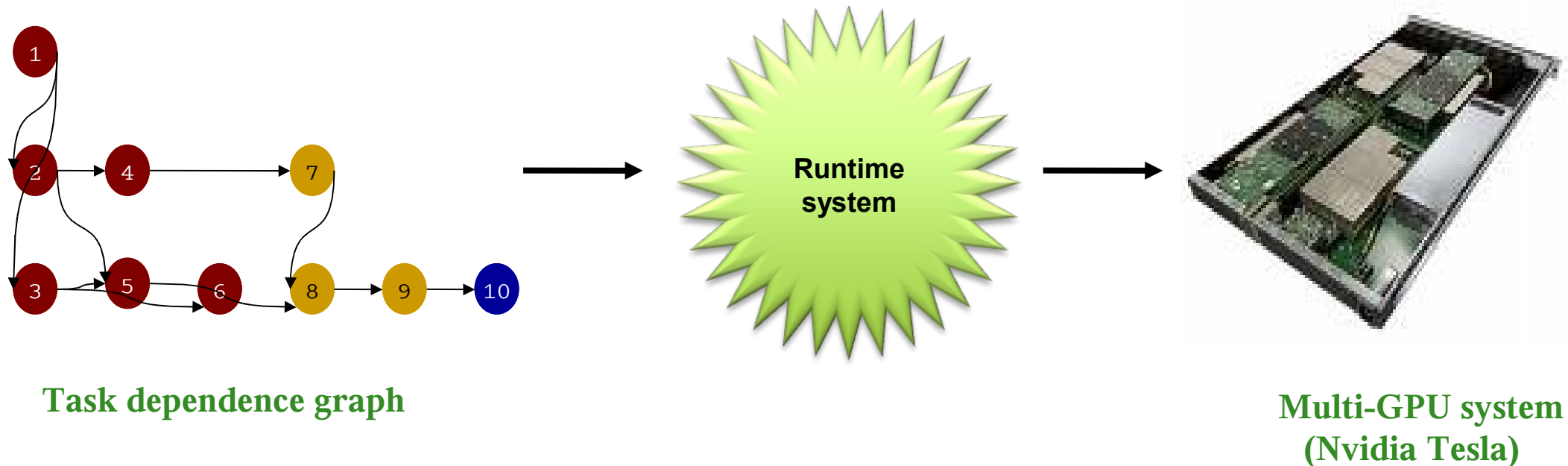


# GPUSs compiler: Linker phase





- Detection of tasks and dependences at run-time
- Generation of graph



- Scheduling of tasks
- Mapping of tasks to GPUs or CPUs
- Data transfers and device memory management
- Exploitation of locality (data transfers reduction)

Sorry, no results or conclusions yet, this is ongoing work!

# Extending the StarSs Programming Model to Platforms with Multiple Hardware Accelerators

**Barcelona Supercomputing  
Center**



**Universidad Jaime I de  
Castellón**



- Files
  - app.c: User code, with GPUSs annotations
  - cellss-gpu-cc\_app.c: specific code generated for the gpu (tasks code written in CUDA)
  - cellss-cpu-cc\_app.c: specific code generated for the cpu (main program)
  - app.tasks: list of annotated tasks
- Compilation steps
  - mcc: source to source compiler, based on the Mercurium compiler (BSC).
  - GPU compiler: CUDA compiler (Nvidia NVCC)
  - CPU compiler: Generic CPU compiler
  - pack: Specific CellSs module that combines objects (BSC)

- Files
  - exec-adapters.c: code generated for each of the annotated tasks to uniformly call them (“stubs”).
  - exec-registration.c: code generated to register the annotated tasks
- Linker steps
  - unpack: unpacks objects
  - glue code generator: from all the \*.tasks files of an application generates a single “adapters” file and a single “registration” file per executable
  - CPU and GPU compilers and linker