UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Compiling parallel code

~~Parallel code compilation~~

Paolo Burgio
paolo.burgio@unimore.it

The most powerful programs

A compiler is a computer program

- ✓ ..that translates source code written in [high-level] programming language
  - C, C++, Java?..
- ✓ ..in lower level language
  - Assembly, CLR?
- ✓ ..(almost) functionally equivalent

Today, we will see C/C++ compilation for parallel programs

- ✓ But first, some background

Processo di traduzione

Programma originario

```
main()
{ int A;
    ...
  A=A+1;
  if ...
```

Programma tradotto

```
00100101
   ...
11001..
1011100 ...
```

- I traduttori convertono il testo dei programmi scritti in un particolare linguaggio di programmazione, **programmi sorgenti**, nella corrispondente rappresentazione in linguaggio macchina, **programmi eseguibili**

Compilation steps

- ✓ Compilation is made of multiple steps
  - Some language(s)-independant, some language(s) dependant
  - We will see C/OpenMP

1. Pre-processing
   - Performs code inclusion and macro expansion

2. "Compilation"
   - "The big guy"
   - Name mismatch

3. Linking
   - Resolves missing (`extern`) symbols and creates executable

1. Pre-processing

- ✓ In C parses (some) pre-processor directives
  - Typically, it performs very stupid code inclusion and replacement
  - (The pre-compiler is NOT stupid, it only does what YOU tell him to do)

```
#include "foo.h"

#define N 11

int main()
{
   int num = N;
   foo(num);

   return 0;
}
```

Let's see this in action

2. "Compilation"

- ✓ Does "the work"
- ✓ Yes, this name generates confusion…
- ✓ That's what we're about today!

3. Linking

- ✓ Resolve missing symbols
- ✓ During "compilation", only symbol names are necessary
  - Not symbol definition

main.c

```c
#include "foo.h"

#define N 11

int main()
{
  int num = N;
  foo(num);

  return 0;
}
```

foo.h

```c
#ifndef __FOO_H__
#define __FOO_H__

extern int foo(int a);

#endif
```

foo.c

```c
#define NUM 5

int foo(int a)
{
  return a + NUM;
}
```

Symbols resolution

- ✓ Then, at linking time we need to resolve the symbol
  - If (global) var, its location (BSS)
  - If function, its address in memory (for call/jumps)

main.o

```
...

jmp foo

...
```

foo.o

```
<foo>:

add r1, r0, 5

jmp r15
```

Let's see this in action

Use only headers?

- ✓ What if I declare everything in headers?
  - **`static`** variables and functions
- ✓ Can include multiple times header file
- ✓ More cumbersome to mantain
  - What if cross-references?

goo.h

```
#include "foo.h"

static int goo()
{
   foo();
}
```

foo.h

```
#include "goo.h"

static int foo()
{
   goo();
}
```

EVERY TIME YOU DO THIS:

goo.c

```c
#include "foo.h"

static int goo()
{
   foo();
}
```

foo.h

```c
extern int goo();

static int foo()
{
   goo();
}
```

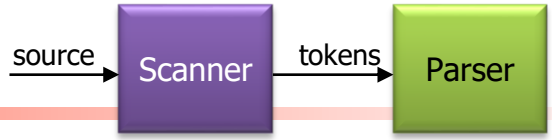
A BUNNY DIES.

"Compilation-compilation"

Typically, three subsystems

- ✓ Frontend: syntax/semantic analysis of source code and creation of internal representation
- ✓ Core: performs transformation/optimization/other..
- ✓ Backend: generates "final" code – we won't see this today
  - ASM/RTL
  - intermediate (Java's Bytecode, .NET's CLR)
  - Source code (Source-to-source compilers)

Frontend



- ✓ Parses pre-processed input file
- ✓ Parsing: reconstruct the derivation (syntactic structure) of a program
  - AKA: program tree

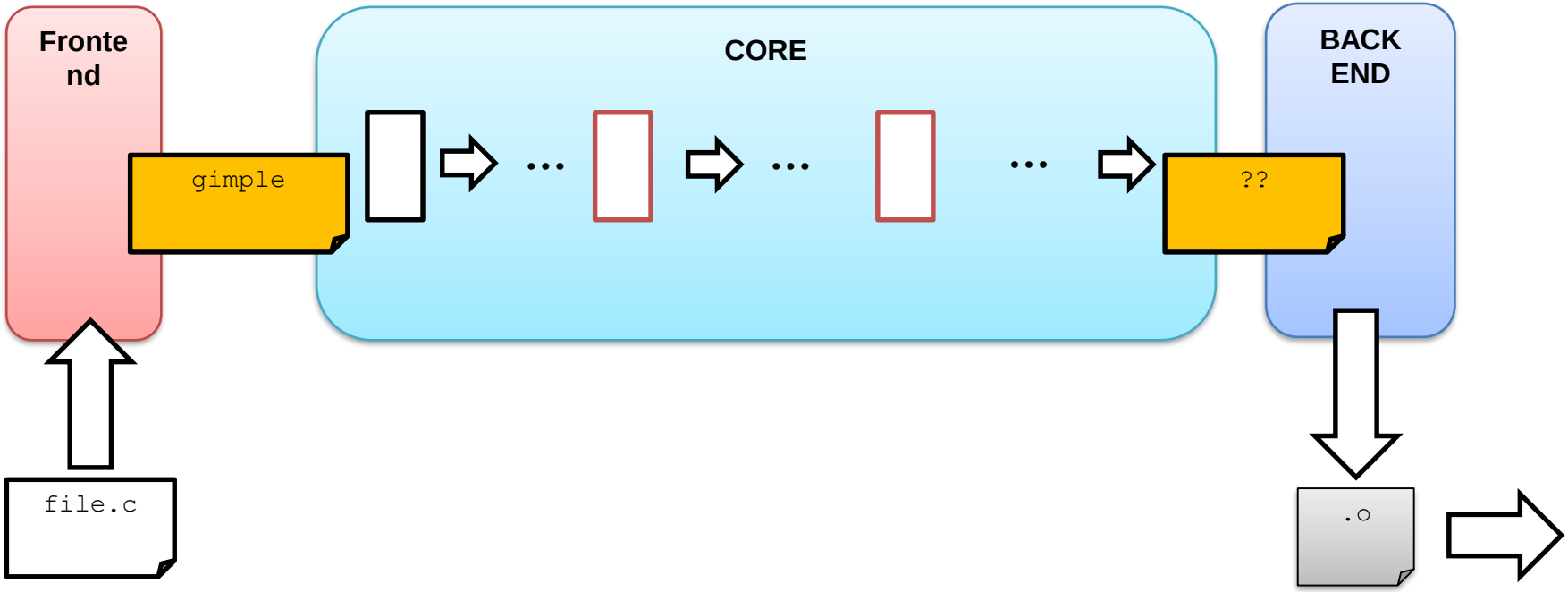
Split in two steps

- ✓ Scanner: translate input characters to tokens
  - Also, report lexical errors like illegal characters and illegal symbols
- ✓ Parser: read token stream and reconstruct the derivation

- ✓ Input text
  - `if (x >= y) y = 42;`
- ✓ Token Stream

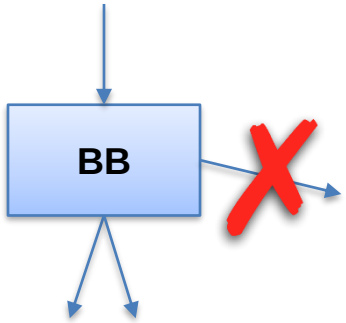
Compilation core

- ✓ Split into steps
- ✓ Reference: GCC
  - Show output from every/all steps: -fdump-tree-*
  - Internal representation called `gimple`



Code representation

- ✓ Based on Basic Blocks
  - *"A straight code sequence with no branches in other than the entry and no branches out other than the exit"*
  - https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html
  - They contain statements (not necessarily instructions)



Typical optimizations #1

- ✓ Removed unused variables
  - Save space in program data

```
void foo()
{
  int a = 11, b = 4;

  printf("a is %d\n", a);
}
```

Typical optimizations #2

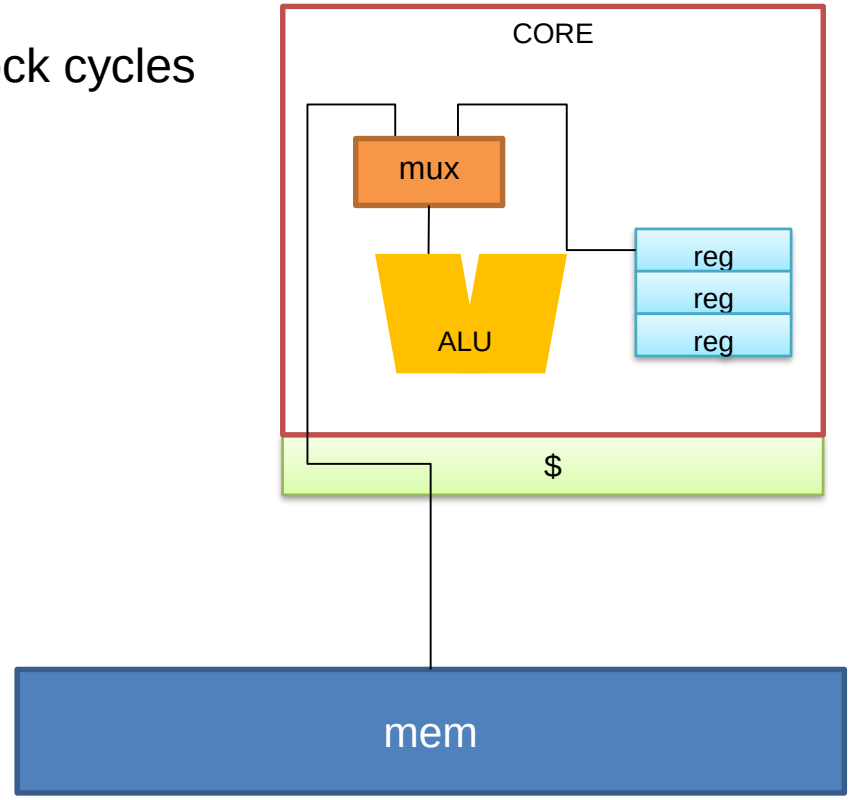
- ✓ Dead code removal
  - Save space in program image

```
int main()
{

  if(false)
    printf("This is not possible\n");
  else
    printf("I will always print this\n");

  return 0;
}
```

Mapping variables

- ✓ Mapping storage in register vs. main memory
- ✓ Different time penalties for accessing a storage
  - Register: 0 (extra) clk cycles
  - Main memory (RAM): ~10s clock cycles
  - If hot D$: 1-2 clock cycles

```
void foo()
{
  int a[16], i;
  for(i=0; i<16; i++)
  {
    // Do something
    a[i] = ...
  }

}
```



Loop unrolling

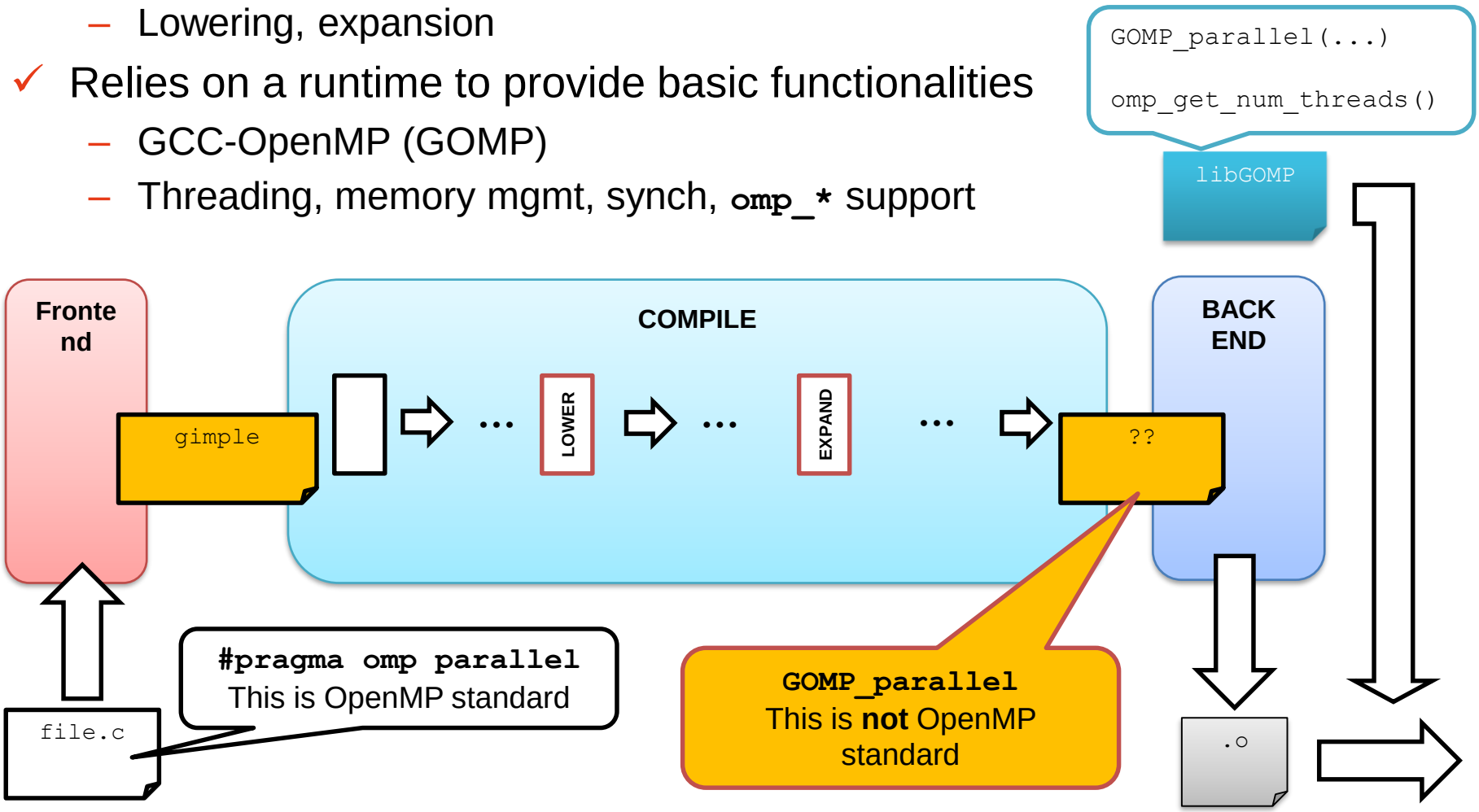
- ✓ Driven by a rolling factor R
  - In this example, R = 4
- ✓ Typically, authomatic
  - Some compilers/tools let programmer specifying this manually
- ✓ Why?

```
void foo()
{
  int a[20], i;
  for(i=0; i<20; i++)
  {
    // Do something
    a[i] = ...
  }

}
```

```
void foo()
{
  int a[20], i;
  for(i=0; i<20; i+=4)
  {
    // Do something
    a[i] = ...
    a[i+1] = ...
    a[i+2] = ...
    a[i+3] = ...
  }
}
```

GCC compiler for OpenMP code

- ✓ Two main steps are performed
  - Lowering, expansion
- ✓ Relies on a runtime to provide basic functionalities
  - GCC-OpenMP (GOMP)
  - Threading, memory mgmt, synch, **`omp_*`** support

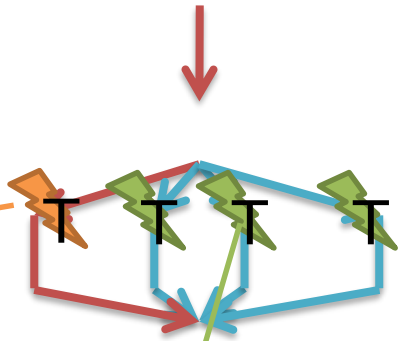


So, the question is

- How do I turn this..

```
int main()
{
  /* Sequential code */
  int num = 11, myid;

  #pragma omp parallel \
    private(myid) shared(num)
  { /* Parallel code */
    myid = omp_get_thread_num();
    foo(11 + myid);
  } /* End of parallel code */


  /* Sequential code */

  return 0;
 }
```

- …into this?




```
int main()
{
  int num = 11, myid;
  // Code to FORK threads
  // Also master thread works!
  myid = omp_get_thread_num();
  foo(11 + myid);

  // Code to JOIN threads

  return 0;
}
```

```
// Wait for something to do

myid = omp_get_thread_num();
foo(11 + myid);

// Notify master that work has ended
```

Parallel region transformation

- ✓ Parallel code is extracted in a separate function
  - Statically/by compiler
  - Expansion
- ✓ So that parallel threads receive a pointer to that function
  - ..and to shared data!
  - Dynamically/at run-time
- ✓ Lowering => Passes to move/prepare data
  - Use of pointers for shared data
  - Data movement at runtime

```
int main()
{
  /* Sequential code */
  int num = 11, myid;

  #pragma omp parallel \
   private(myid) shared(num)
  {/* Parallel code */
   myid = omp_get_thread_num();
   foo(11 + myid);
  } /* End of parallel code */


  /* Sequential code */

  return 0;
 }
```

Let's see this in action

A parallel software stack

Application(s)

Hooks (APIs)

OpenMP

Parallel framework

OS hooks (APIs)

pthreads

Java

C++ new/delete

C malloc/free

Middle level/services API

OS level

Hardware level + BSP

OpenMP software stack

Applications

OpenMP API

- Compiler directives (Pragmas)
- Runtime library routines
- Environmental variables

OpenMP runtime

- Parallel threading
- OMP memory mgmt
- OMP synchronization

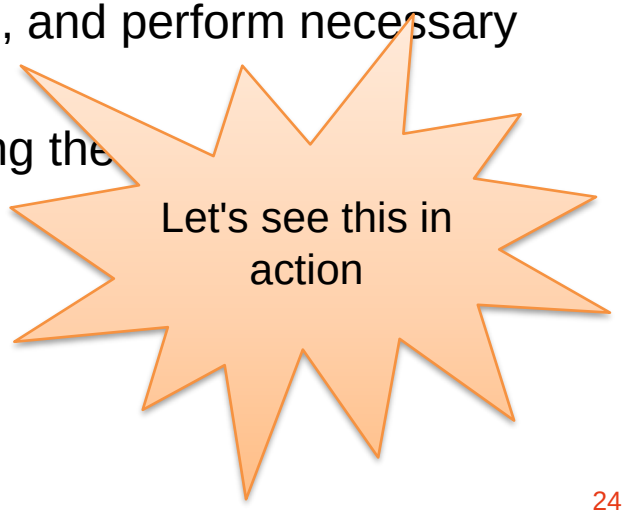
Service API

- Basic threading library
- (Shared)memory mgmt
- Synchronization

Hardware+OS level

What happens at runtime

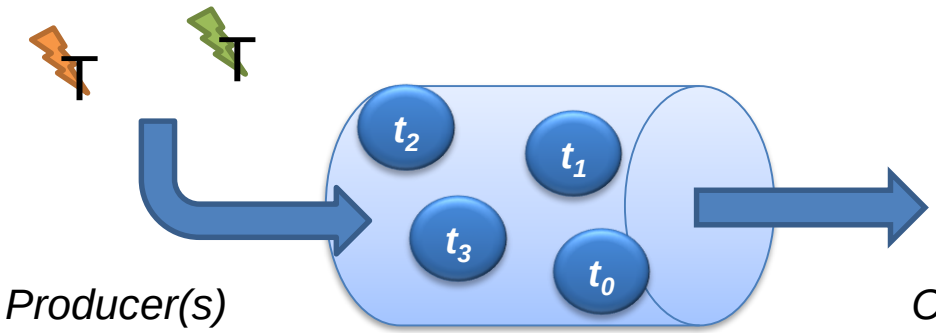
- ✓ See GOMP runtime internals
- ✓ For each pragma, there is a coresponding `GOMP_` function
  - One, or more than one
- ✓ `GOMP_parallel` is in charge of
  - Computing exact number of threads required
  - Creating a team of N-1 threads
    - Leverages on OS threading, or other APIs (PThreads)
  - Taking data pointers (lowered at compile time), and perform necessary data movements and initialization
  - Checking when threads are done, and releasing the
    - Performs join



Another example: tasks

- ✓ `GOMP_task` to insert in the pool
- ✓ Task scheduling points are at boundaries of `GOMP_*` calls
  - Can be implemented inside runtime
  - No need to modify code to express them

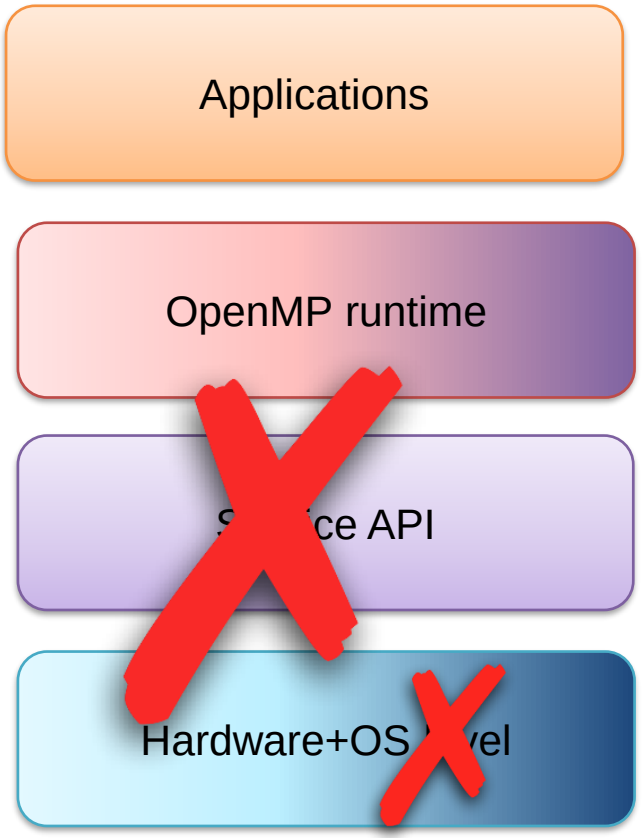
```
/* Create threads */
#pragma omp parallel num_treads(2)
{
  /* Push a task in the q */
  #pragma omp task
  {
    t0();
  }

  /* Push another task in the q */
  #pragma omp task
    t1();

} // Implicit barrier
```



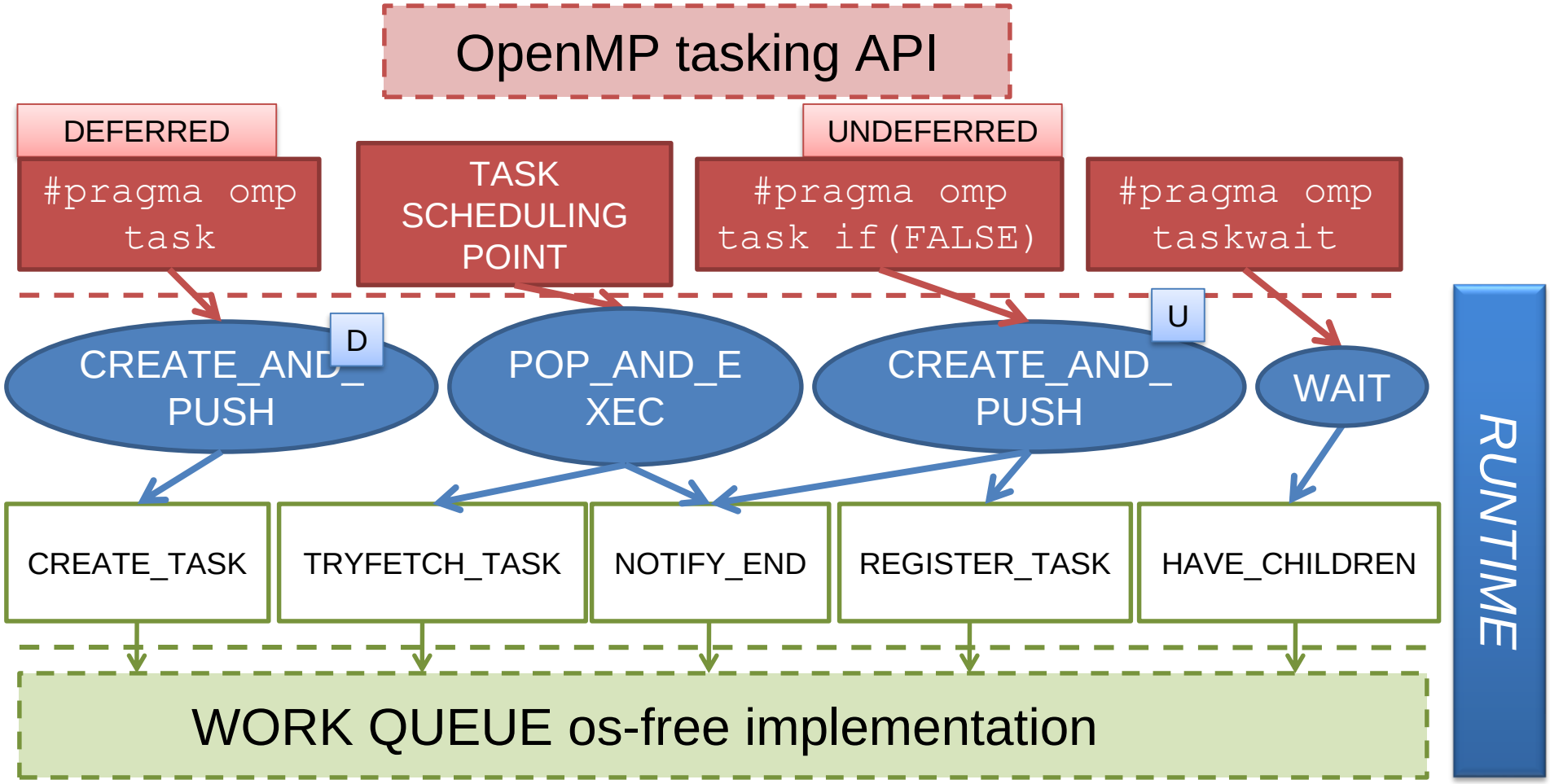
Let's see this in action

Highly-optimized runtime

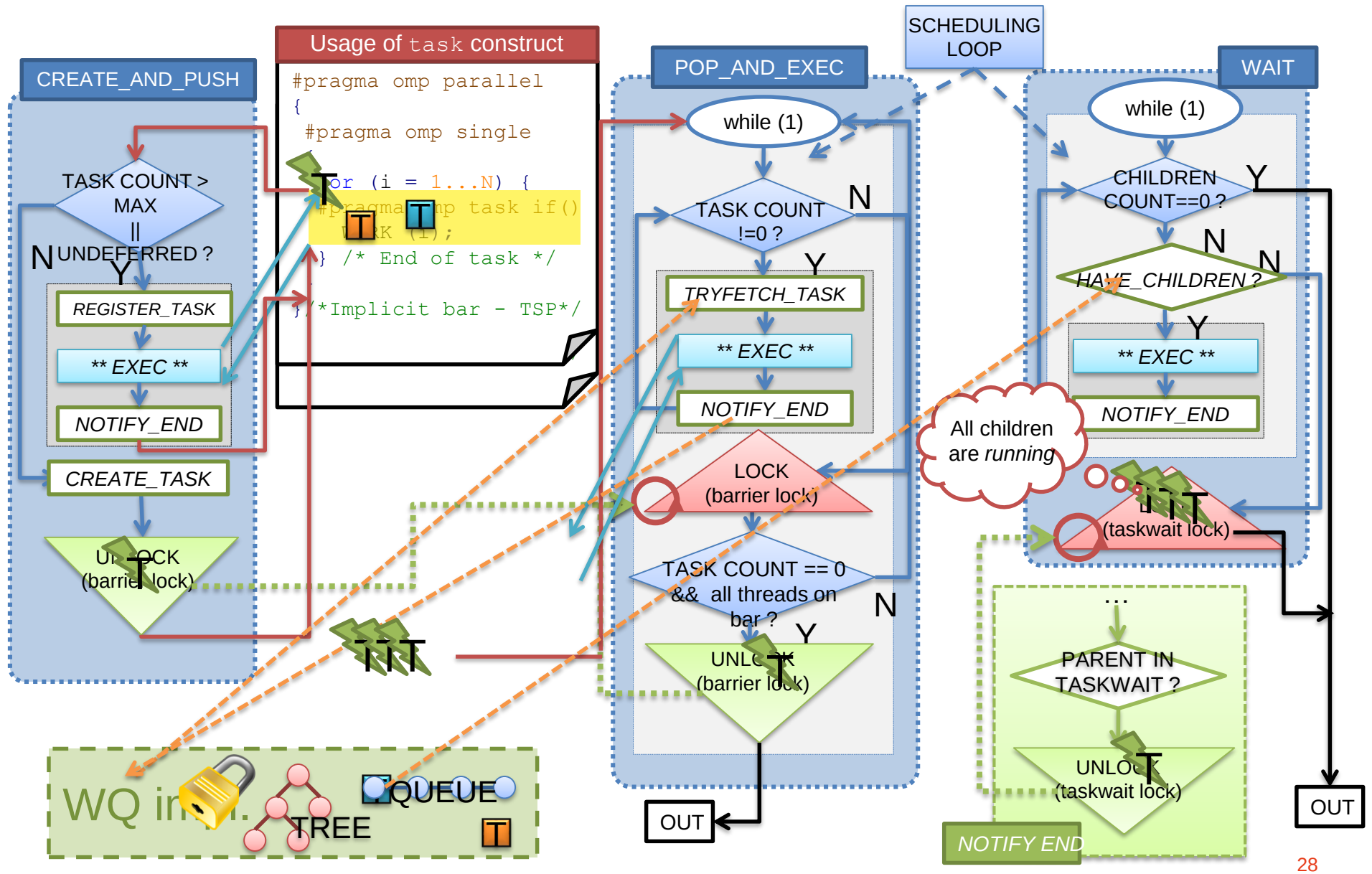
- ✓ Service layer has a cost!
  - Might want to remove, e.g., PThreads
- ✓ OS might be absent!
  - Run directly on BSP
- ✓ Typical of embedded systems
- ✓ Runtime development might become a nightmare
  - ..but worth for performance
  - Up to 100x



Highly-optimized runtime

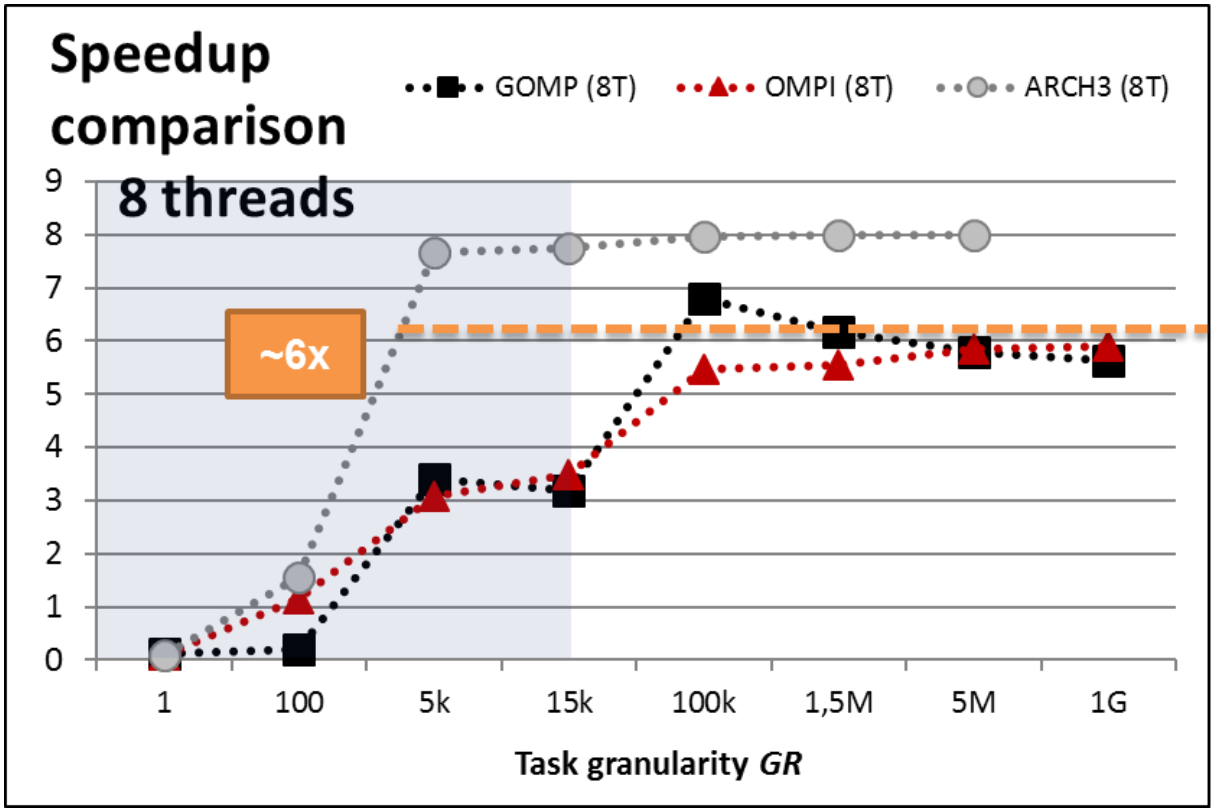


An extremely complex runtime…



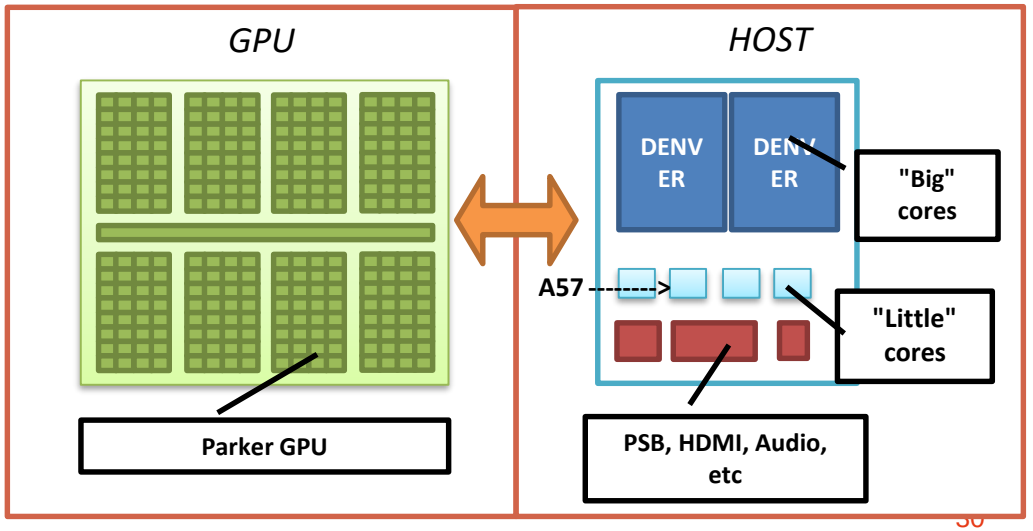
…but worth

- ✓ Ideal speedup for granularities 20times smaller!



Compiling for heterogeneity

- ✓ Example: architectures with a host and an accelerator
- ✓ Example: NVIDIA Tegra X2
  - ARM quad-core host
  - NVIDIA Parker GPU w /hundred cores
- ✓ The problem is, might have different ISAs!!
  - Need two backend
  - one for the ARM ASM
  - and one for the GPU "ptx" ASM



A small recap

- ✓ Write code for host..and for device
  - They can also be in the same file..

host-code.cu

```
int main()
{
  printf("[HOST] Hello World!\n");

  funzione<<<3,5>>>();

  cudaDeviceSynchronize();

  printf("[HOST] Device ended its work!\n");

  return 0;
}
```

device-code.cu

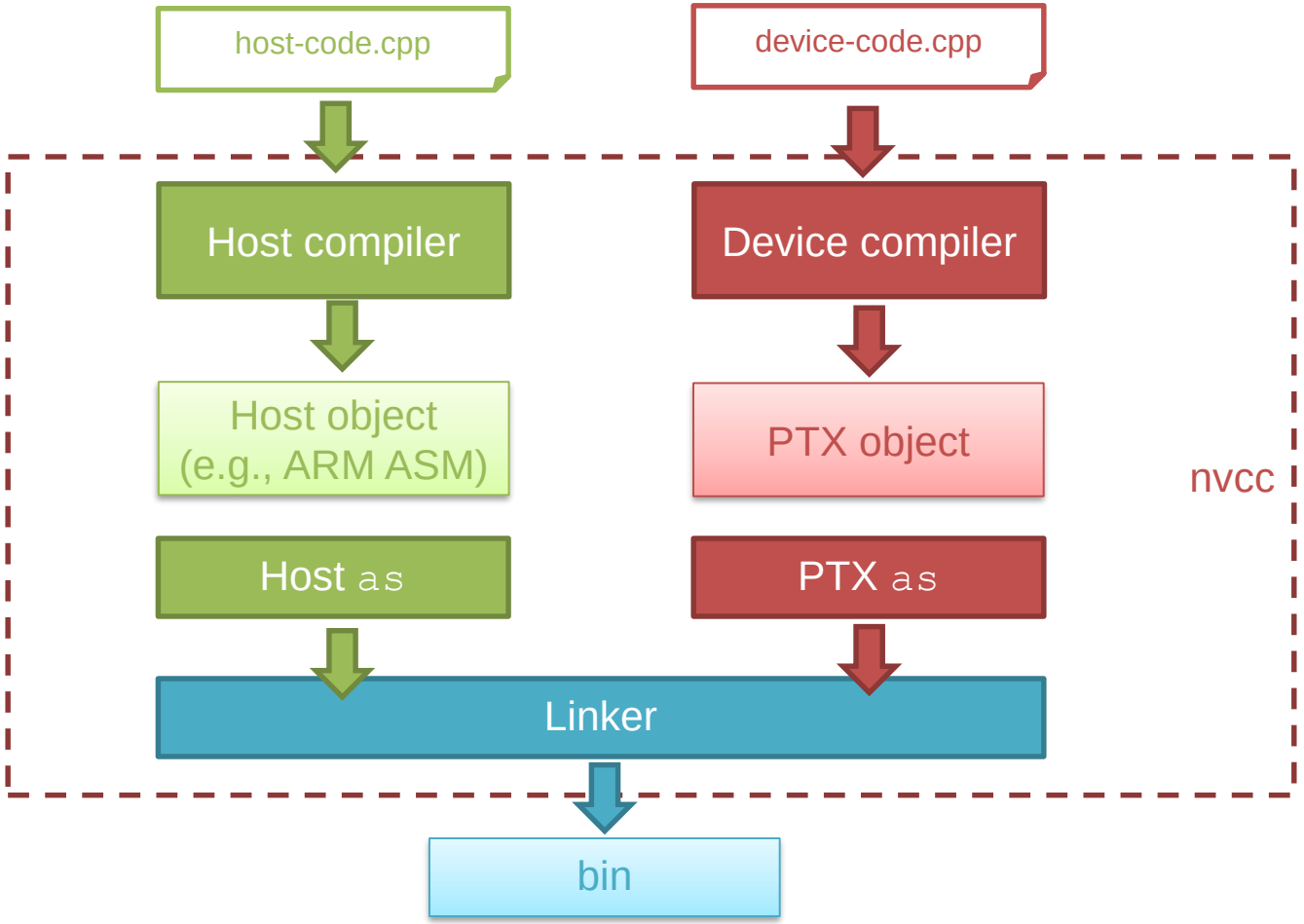
```
#include <stdio.h>
#include <cuda.h>
__global__ void funzione()
{
  int thrId = threadIdx.x;
  int blkId = blockIdx.x;
  int thrNum = blockDim.x;
  int blkNum = gridDim.x;
  printf("\t\t\t\t\t[DEVICE] Hello World! \
          I am thread #%d out of %d, and I belong\
          to block #%d out of %d\n",
          thrId, thrNum, blkId, blkNum);
return;
}
```

How would you do that?

- ✓ Move/extract kernel code
  - In CUDA, we can ask programmers to do it
  - Using `_device_/_global_` keywords to identify code that runs on GPU
- ✓ Then, compile separately
  - And (optionally) pack in a unique file
- ✓ CUDA runtime/drivers/firmware take care of everything
  - "Closed" and proprietary

Compilation scheme

- ✓ Assume we have separate host/gpu source files



A programmability issue

- ✓ Programmers must write code in "one language and a half"
  - CUDA is an extension of C++
- ✓ Since Spec 4.5, OpenMP has "elegant and appealing" extension for hetereogeneity
  - `#pragma omp target`
- ✓ We will see how to do it starting from OpenMP
  - Generate CUDA code
  - Use CUDA as (opaque) Service Layer
  - Hercules H2020 project

OpenMP 4.5

```
#pragma omp target [clause [[,]clause]...] new-line
  structured-block
```

Where clauses can be:

```
if([ target :] scalar-expression)
device(integer-expression)
private(list)
firstprivate(list)
map([[map-type-modifier[,]] map-type: ] list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

- ✓ OpenMP 4.5 introduces the concept of device
  - Execute `structured` block onto `device`
  - `map` clause to move data to-from the device

The previous program

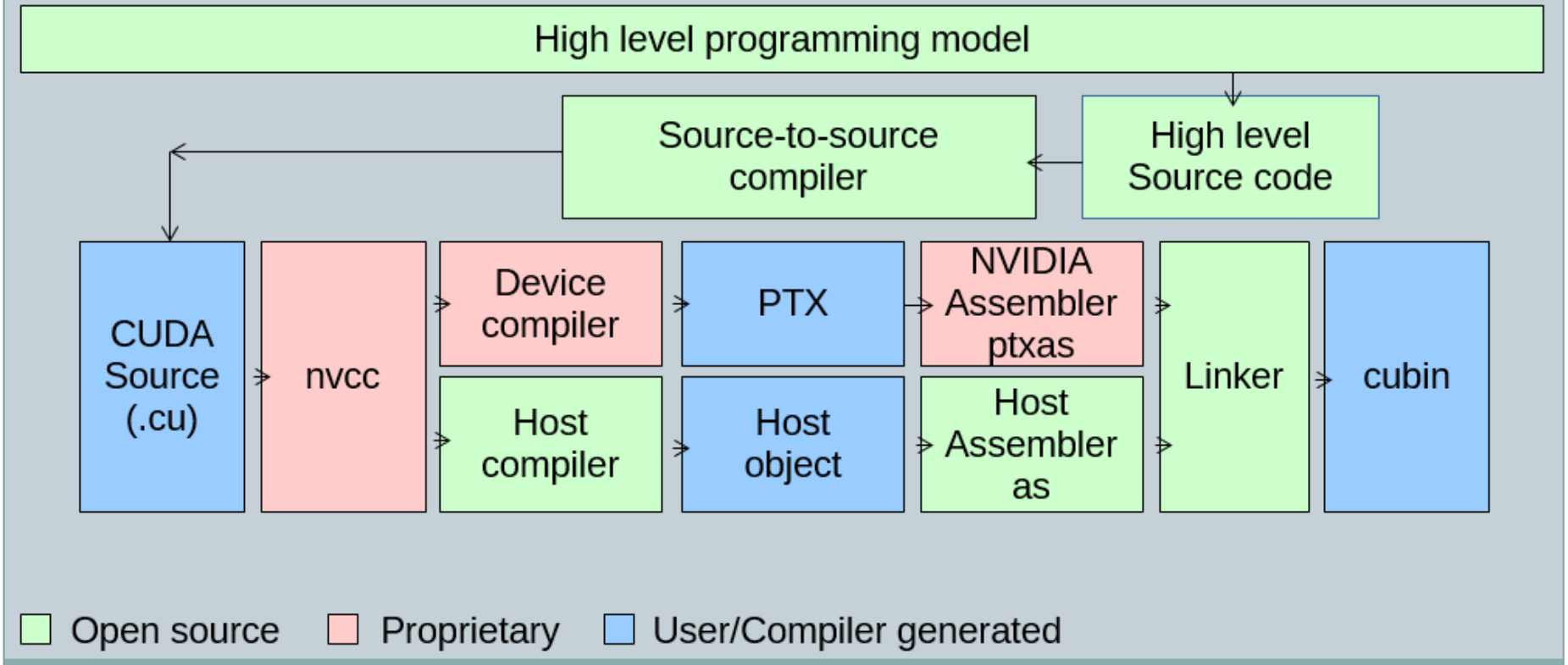
```c
int main()
{
  printf("[HOST] Hello World!\n");

  funzione<<<3,5>>>();

  cudaDeviceSynchronize();

  printf("[HOST] Device ended its work!\n");

  return 0;
}
```

```c
#include <stdio.h>
#include <cuda.h>
__global__ void funzione()
{
  int thrId = threadIdx.x;
  int blkId = blockIdx.x;
  int thrNum = blockDim.x;
  int blkNum = gridDim.x;
  printf("\t\t\t\t\t[DEVICE] Hello World! \
          I am thread #%d out of %d, and I belong\
          to block #%d out of %d\n",
          thrId, thrNum, blkId, blkNum);
  return;
}
```

```c
int main()
{
  printf("[HOST] Hello World!\n");

  #pragma omp target
  {
     printf("\t\t\t\t\t[DEVICE] Hello World!\n");
  }

  printf("[HOST] Device ended its work!\n");
  return 0;

}
```

Source to source compilation

- Use source-to-source compiler to transform a high-level program specification into a CUDA program



Source-to-source compilers

- Compilers that as a backend produce some other type of code
- Remember `-fdump-tree-all`?
  - Produces gimple

References

- "Calcolo parallelo" website
  - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- OpenMP specifications
  - http://www.openmp.org
- CUDA specifications & dev. Toolkit
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/
  - http://docs.nvidia.com/cuda/#axzz4dGU41K8e