modelica2GPU

Index

Qualche nozione iniziale

Sempre più, nella nostra realtà, sta diventando importante modellare e simulare sistemi cyber-fisici, come droni o sistemi IoT, e sistemi biologici, come il ciclo cellulare o ancora lo sviluppo di un virus (ogni riferimento a fatti o virus è puramente casuale). Sebbene questi due siano argomenti diversi, hanno principalmente una cosa in comune: entrambi possono essere espressi come un sistema di equazioni differenziali ordinarie (ODE) il quale può essere integrato sul tempo al fine di dare una "vita" al modello stesso il quale risulterà in un dataset con cui si andrà a descrivere l'andamento, in termini di stati, quindi la traiettoria, del sistema stesso nel tempo. Questa fase di integrazione viene chiamata *simulazione* del modello, la quale ovviamente non comprende solo l'operazione di integrazione ma anche tante altre cose tra le quali il riconoscimento degli **eventi**.

> Gli **eventi** sono delle situazioni alternative che deviano il normale corso della traiettoria del sistema, di conseguenza esistono due tipologie di eventi: di stato e di tempo. I primi sorgono da *condizioni* sugli stati, ad esempio `x > 10 and x < 20` , mentre i secondi da condizioni sul tempo. Su quest'ultimi il discorso è più ampio, però generalizzando molto si potrebbe dire che questi servano principalmente per immettere una componente discreta all'interno di un sistema continuo, il più classico è il **sample&holding**. Volendo fare un parallelismo con la CPU (la quale è ovviamente una componente continua), possiamo dire che ogni colpo di clock (= 1) e tutto ciò che avviene successivamente è un evento sul tempo.

Ovviamente ogni sistema, se descritto da ODEs, deve avere delle equazioni iniziali per ogni parametro/variabile coinvolta nel sistema di equazioni differenziali altrimenti il sistema si dice **under-determined**, e alcune equazioni iniziali possono cambiare da simulazione a simulazione. Infine un sistema, che sia biologico o cyber-fisico con tutte le componenti appena descritte, può essere implementato tramite il liguaggio di modellazione Modelica, il quale è costruito adhoc per sistemi elettrici in quanto presenta di default molti componenti come Resistor e altri, ma si è visto che può essere utilizzato per descrivere un qualsiasi modello continuo o discreto che sia, anche sistemi biologici che in nativo sono espressi con SBML.

> **SBML**, o System Biology Markup Language, è un particolare formato di rappresentazione, molto simile se non uguale a XML per la descrizione di modelli computazionali di processi biologici. Esiste un tool per la costruzione di file Modelica partendo da SBML, che si chiama sbml2modelica.

Modelica quindi mette a disposizione tutto il necessario di un linguaggio di programmazione orientato alla modellazione di sistemi continui o non, fornendo la possibilità di definire: variabili (Real, Boolean, Integer, String), anche con valori di inizio, e tipi user-defined che estendono quelli primitivi; equazioni iniziali sotto la keyword *initial equation*; equazioni differenziali e alegbriche, sotto la keyword *equation*; eventi, espressi tramite le cosìddette **when equation** dal momento che matchano un blocco `when <condition> then; ... end when;` ; assegnazioni, ossia equazioni algebriche ma che sono presenti sotto la keywork *algorithm*.

> *Quale differenza c'è tra le equazioni algebriche sotto equation e quelle sotto algorithm*? In Modelica tutto ciò che è presente nel blocco *equation* viene eseguito in un vero e proprio parallelismo, quindi non concorrenza, in quanto le diverse equazioni accedono sì alle stesse risorse (variabili), che quindi sono condivise, ma quest'ultime non vanno ad essere aggiornate nello stesso blocco altrimenti il parallelismo diverrebbe concorrenza. Di conseguenza queste variabili vengono aggiornate all'interno del blocco *algorithm* in quanto, proprio come gli algoritmi, vengono eseguiti dall'alto al basso così come sono scritti (proprio per questo si chiamano assegnamenti). Questa differenza sarà molto importante per il traduttore.

Una volta che abbiamo tutto in codice Modelica, le sue librerie offrono software e strumenti di simulazione, ma che di base utilizzano la CPU. Questa può essere molto performante su singoli test, ma ovviamente nel momento in cui vogliamo simulare un sistema più volte con input diversi non lo possiamo fare sequenzialmente e quindi dobbiamo ricorrere al multithreading. Come è ben noto la computazione multithreading è molto più performante quando eseguita su GPU piuttosto che su CPU, la quale a sua volta potrebbe inceve eccellere su singole (o poche in paralello) simulazioni. Un framework che permette di fare questo è MPGOS, o Massively Parallel GPU-ODE Solver, il quale prende in input modelli in C++, descritti secondo uno specifico format, compila con *nvcc*, o nVidia CUDA Compiler, producento in output un .exe da runnare, il quale farà partire un tot di simulazioni con tutte le configurazioni che gli sono state date durante la costruizione dei due file necessari al run: ossia il file .cu ed il file _SystemDefinition.cuh, i quali descrivono rispettivamente i settings per la simulazione (numero di thread attivi, numero di thread per blocco, la GPU da utilizzare, le equazioni iniziali, ...) ed il sistema stesso (ODEs, eventi, ...).

Finalmente arriviamo a noi ...

modelica2GPU è un tool di traduzione automatica da codice Modelica a codice C++ già formattato e pronto per la compilazione e l'esecuzione tramite il framework MPGOS. *Perché è necessario*? La risposta è semplice, quando si tratta con modelli molto lunghi e quindi con tante equazioni differenziali, iniziali e tant'altre configurazioni da settare in uno dei due file, è molto comune cadere in qualche errore di trascrizione del modello sia per le motivazioni descritte precedentemente sia per un misunderstanding su quale setting da impostare per far funzionare le cose. Automatizzando il lavoro, si risparmi all'utente finale, il compito tedioso e di certo non privo di errori, della trascrizione del modello. Di certo il fine di chi utilizza MPGOS non è tanto capire come funziona la programmazione su GPU ma tanto quello di prendere un modello, buttarcelo dentro e vedere i risultati delle simulazioni, senza stare troppo a ragionare su come settare determinate impostazioni che invece potrebbero essere impostate di default. modelica2GPU permette di levare all'utente finale un compito del genere a scapito della costruzione di un semplicissimo file di configurazione in formato yaml il quale dovrà poi essere dato in pasto al traduttore il quale si occuperà di tutto il resto. Nel file di configurazione si dovranno impostare dei valori come: + **working directory**: la directory in cui sono presenti i Modelica ed in cui si andrà a salvare l'output. + **model name**: Il nome del file Modelica principale, che connette tutte le altre componenti. + **xml**: Il path assoluto al file XML che descrive il modello. + **logger**: True se si vuole loggare su file, False altrimenti + **notifier**: True se si vogliono abilitare le notifiche, False altrimenti + ... altre configurazioni

> Notiamo come tra le configurazioni ci sia il campo **xml**, questo perché modelica2GPU prende sì i modelli Modelica, ma tramite il compilatore omc (OpenModelica Compiler), va a generare un XML che crea una versione *flatten* del sistema, ossia in cui tutte le equazioni sono già interlacciate tra di loro e ottimizzate. L'uso dell'XML, data la formazione del modello flatten, semplifica di molto il parsing degli elementi oltre a dare anche una forte nota di generalizzazione tra i diversi modelli la cui struttura XML seguirà pattern ben specifici imposti dagli schemi XSD ai quali fanno riferimento. C'è da dire putroppo che alcuni sistemi non sono stati tradotti bene dal compilatore generando degli errori importanti. Ma delle limitazioni di modelica2GPU ne parleremo nelle prossime sezioni.

Ovviamente dovranno essere settati anche valori di MPGOS, molti, ai quali però potrà essere dato un valore di default semplicemente mettendo a True l'opzione *usedefaultoptions* nel file di configurazione.

> Ovviamente, vorrei far notare, che tali valori di default possono essere modificati andando nel file **modelica2GPU.py** sotto la directory *src* e più precisamente nella funzione *getdefaultoptions*.

modelica2GPU Workflows

In generale il workflows che modelica2GPU segue è il seguente:

1. Parsing della configurazione dal file di configurazione dato in input
2. Setting della configurazione
    1. Nel caso in cui non ci sia l'opzione xml, si va a generare l'XML tramite compilazione dei Modelica
3. Costruzione del parser dell'XML
4. Parsing delle informazioni necessarie
    1. Parsing delle variabili
    2. Costruzione dei parametri MPGOS (X, ACC, ACCi, sPAR, sPARi) dalle variabili parsate
    3. Parsing delle equazioni iniziali e aggioranamento delle var con i nuovi valori iniziali
    4. Parsing delle funzioni user-defined e associazione con quelle built-in
    5. Parsing delle equazioni dinamiche (ODE, algeb ed eventi)
    6. Parsing degli algoritmi
5. Costruzione dell'astrazione del modello in studio tramite struttura dati astratta (classe)
    1. Possibile utilizzo di ordinamento topologico per ordinare le equazioni iniziali
6. Costruzione del builder dei file per MPGOS
    1. Costruzione del SystemDefinitionBuilder per la costruzione del file .cuh
    2. Costruzione del ModelDefinitionBuilder per la costruzione del file .cu
    3. Costruzione del makefile
7. Creazione dei file cuh, cu e makefile

Installazione e dipendenze

modelica2GPU non ha una vera e propria installazione, in quanto anche in questo caso è già stato tutto fatto. Infatti il software viene distribuito già con l'eseguibile creato e salvato nella directory *build* (sta allo stesso livello della directory *src*). Tale file è stato creato con **pyinstaller**, un modulo python, istallabile con *pip*, il quale dato in input un file genera l'eseguibile con il suo stesso nome. La cosa interessante di *pyinstaller* è che generando l'exe incapsula in esso tutte le librerie necessarie al corretto funzionamento di modelica2GPU, permettendo all'utente finale di non dover installare moduli python aggiuntivi come + **notify2** per le notifiche + **pycuda** per la gestione delle GPU nVidia + **coloredlogs** per la possibilità di avere i log colorati su terminale. + **colorama** il quale viene utilizzato insieme a quello precedente, da parte di Windows

> Ovviamente tutti installabili con *pip install*

Purtroppo alcune dipendenze devono essere necessariamente installate o possedute, queste sono: + Macchina con una GPU nVidia che permettono l'utilizzo del CUDA toolkit + Il CUDA toolkit (versione >= 11), con il compilatore nvcc. Visitare CUDA download site + OpenModelica Compiler, per creare l'XML. Scaricare da qui + Il framework MPGOS, scaricabile da qui

Opzionalmente è possibile installare + Modulo python *pyinstaller* tramite pip, nel caso in cui si voglia modificare il codice sorgente e ricompilare

---

File di configurazione

modelica2GPU prende in input due file e uno dei due è il sopracitato *file di configurazione*. Come abbiamo già detto, esso è un file molto importante dal momento che detterà il 90% della formattazione dei due file per MPGOS dati in output, ma soprattutto imposta i parametri per la gestione delle simulazioni. Il file di configurazione è un formato **yaml**, non molto diverso dal **json**, con l'unica differenza che non ci vanno gli apici: le parole quindi saranno interpretate come stringhe, i numeri come numeri (int, float, ...), i True e False come booleani e i null come i None (in Python). Molto importante è che i campi andranno a matchare i tipi corrispondenti, che saranno esplicitati nella tabella seguente. Tutti i campi devono essere esplicitati con un valore, oppure con null in base a determinato condizioni.

Campo	Descrizione	Tipo
modelica2GPU	Super-campo, descrive l'inizio della della configurazione per il tool	Nessuno
generateXML	Indica se si vuole generare l'XML tramite il compilatore Modelica	Boolean
omlibrary	Path assoluto alla libreria Modelica	String
xml	Path assoluato all'XML del modello in studio, null se generateXML è True	String
workingdir	Path assoluto alla directory con i Modelica e nella quale si vuole salvare l'output	String
modelfilename	Filename del file Modelica che descrive l'intero modello	String
notifier	Indica se si vogliono attivare le notifiche di sistema	Boolean
filelogger	Indica se si vuole attivare il logging su file	Boolean
builder	Super-campo, descrive l'inizio della configurazione per il builder dei files	Nessuno
MPGOSsourcedir	Path assoluto alla directory SourceCode di MPGOS	String

Campo	Descrizione	Tipo
usedefaultoptions	Indica se si vogliono utilizzare le opzioni di default oppure se si vuole impostare a mano	Boolean
builder/gpu	Super-campo, descrive l'inizio della configurazione della GPU da utilizzare	Nessuno
major	Il valore Major della CUDA compute capability della GPU alla quale si fa riferimento	Int
minor	Il valore Minor della CUDA compute capability della GPU alla quale si fa riferimento	Int
builder/modeldefinition	Super-campo, descrive l'inizio della configurazione del file .cu	Nessuno
numberOfThreads	Indica il numero di thread che si vogliono riservare al tool - **10000 default**	Int
numberOfProblems	Indica il numero di problemi da risolvere (uno per thread in base ad MPGOS) - **20000 default**	Int
numberOfDenseOutput	Indica il numero di DenseOutput da salvare (da 0 al numero di Thread) - **10000 default**	Int
threadsPerBlock	Indica il numero di thread residenti in un blocco. Dal momento che questo settaggio ha un grande impatto nelle performance della simulazione, allora per default il valore è **32**. Valori più alti dipendono dalle specifiche della GPU	Int
initialTimeStep	TimeStep iniziale della fase di integrazione - **1.0e-2 default**	Float
preferSharedMemory	Indica se dove si vogliono memorizzare i parametri condivisi (sPAR e sPARi): (1) Shared Memory; (0) Global memory. Tenere bene a mente che memorizzare in (1) ridurrebbe molto il tempo di accesso a tali risorse - **1 default**	{0, 1}
maximumTimeStep	Il TimeStep massimo durante una fase di integrazione - **1.0e+6 default**	Float
minimumTimeStep	Il TimeStep minimo durante una fase di integrazione - **1.0e-14**	Float
timeStepGrowLimit	Il tasso massimo di crescita di un TimeStep in caso di uno step accettato (successful) **5.0 default**	Float
timeStepShrinkLimit	Il limite minimo di riduzione di un TimeStep in caso di uno step rifiutato **0.2 default**	Float
eventDirection	Falso super-campo in quanto è null se usedefaultoptions = True. Definisce Il setting per gli eventDirection. Se supponiamo essere N allora a mano dovremmo inserire ... vedi dopo	Nessuno
eventDirectionValue0	Indica se si vuole fare la detection degli evento 0 con tangente negativa (-1), positiva (1) o entrambe le direzioni **(0, default)**	{-1, 1, 0}
...	...	...
eventDirectionValueN	...	...
denseOutputMinimumTimeStep	Indica il minimo tempo di attesa tra step di salvataggio, per prevenire la saturazione dello storage. Ovviamente 0.0 indica che ogni timestep con successo sia registrato - **0.0 default**	Float
denseOutputSaveFrequency	Indica la frequenza, in termini di Step, di salvataggio del DenseOutput.**1, default,** indica ogni timestep avvenuto con successo	Int
tolerance	Falso super-campo in quanto è null se usedefaultoptions = True. Definisce Il setting per le tolleranze sia relative che assolute nel valore degli stati. Se supponiamo essere N allora a mano dovremmo inserire ... vedi dopo	Nessuno
toleranceValue1	Tolleranza relativa e assoluta per lo stato 1 - **deafault preso dall'XML**	Float
...	...	...
toleranceValueN	...	...
timeDomainInit	Istante di tempo in cui inizia la simulazione - **0.0 default**	Float
timeDomainEnd	Istante di tempo in cui termina la simulazione (detto orizzonte) - **10.0 default**	Float

Runnare il tool

Come abbiamo giò detto grazie al modulo pyinstaller è stato possibile creare un file exe (144Mb) chiamato *modelica2GPU.exe* presente nella directory *bin*, il quale potrà poi essere spostato dove si vuole, tanto tutte le dipendeze sono contenute nell'eseguibile stesso. La sintassi per richiamare il tool da linea di comando (l'unica possibilità di eseguirlo) è la seguente

```
./modelica2GPU [-h] -l/--logger LOGGER -c/--config CONFIG
```

dove: - l'opzione -l, --logger serve per inserire il file di configurazione del logger il quale è presente nella cartella *modelica2MPGOSgpuPython/src/config/logger.yaml* - l'opzione -c, --config serve per inserire il file di configurazione del configuratore. Un esempio è presente nella cartella *modelica2MPGOSgpuPython/src/config/modelica2GPu.yaml*

Configurazione delle cartelle

- *modelica2MPGOSgpuPython/*
  - *src/*
    - *builder/*
      - builder.py (Contiene le classi per la costruzione dei file)
    - *config/*
      - logger.yaml (Coontiene le configurazioni per tutti i logger, inizia da solamente con quella del logger su console)
      - modelica2GPU.yaml (Esempio di file di configurazione del traduttore)
    - *exceptions/*
      - builtExceptions.py (Contiene le classi che definiscono le eccezzioni create ad hoc per il traduttore)
    - *icons/*
      - icons8-error-64.png (Icona di errore per il notificatore)
      - icons8-video-card-100.png (Icona del tool per il notificatore)
    - *model/*
      - model.py (Contiene la classe per la definizione dell'astrazione del modello)
    - *parser/*
      - parser.py (Contiene la classe per la definizione del parser dell'XML)
    - *tagclasses/*
      - tagclasses.py (Contiene le classi che servono per mantenere in memoria tutti Tag, infatti le classi fanno proprio riferimento ad essi. Inoltre sono presenti le fuzioni di parsing delle equazioni, funzioni ed eventi)
      - variables.py (Contiene le classe che descrivono i tag per le variabili e tutti i parametri MPGOS)
    - *utils/*
      - notifier.py (Classe per il notificatore)
      - logger.py (Classe per il logger)
      - graph.py (Classe per il DAG per l'ordinamento topologico)
    - *XMLs/*
      - BIOMD0000000005.xml
      - BouncinBall.xml
      - Goldbeter1995.xml
      - MyModel.xml
      - Zeilinger2006_PRR7_PRR9_Y.xml
    - modelica2GPU.py (File main per l'esecuzione del traduttore, richiama tutti i moduli in seguenza come descritto nella sezione precedente)
  - *bin/*
    - modelica2GPU.exe (Eseguibile)
  - README.md (Il file README)

Un esempio di esecuzione secondo la configurazione delle directory precedente.

```
./bin/modelica2GPU --logger src/config/logger.yaml --config src/config/modelica2GPU.yaml
```

Limitazioni temporanee di modelica2GPU

Le limitazioni principali riguardano più che altro quali costrutti Modelica non vengono parsati da modelica2GPU. Per esempio costrutti complessi come interi blocchi If...Then...Else If...Then...Else... ancora non sono implementati, a differenza dei blocchi IfThenElse inline che invece sono presi in considerazione. Alcune funzioni built-in che generano eventi non sono state implementate: abbiamo per esempio il classico *sample*, *reinit*, ma altre come *noEvent* ancora non ci sono. Inoltre in Modelica è possibile definire Array e Matrici, che ovviamente non potranno (almeno per adesso) esser parsate, quindi tutte le variabili devono essere Real, Integer, Boolean, String e tipi derivati. Altra limitazione riguarda i loop i quali non ci devono essere e le funzioni definite dall'utente devono avere solamente un output e l'algoritmo che le descrive deve essere fatto di soli assegnamenti, questo perché il parser degli algoritmi non permette il parsing di algoritmi con blocchi when, if e loop. Ultima cosa, per adesso il builder non permette, qual'ora siano state parsate, la scrittura della definizione di user-defined functions ma andrà ad utilizzare solamente quelle builtin, ossia quelle di cui esiste una classe nella file tagclasses. Ovviamente ultima limitazione riguarda i Tag implementati ... *Sono stati implementati tutti*? Non lo so.