Simulation of Modelica Models on the CUDA Parallel Architecture

Per Östlund

Department of Computer and Information Science
Linköpings universitet

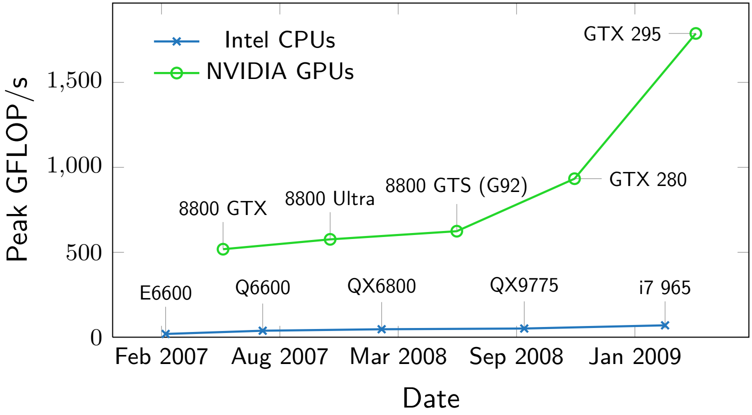
February 7, 2010

Outline

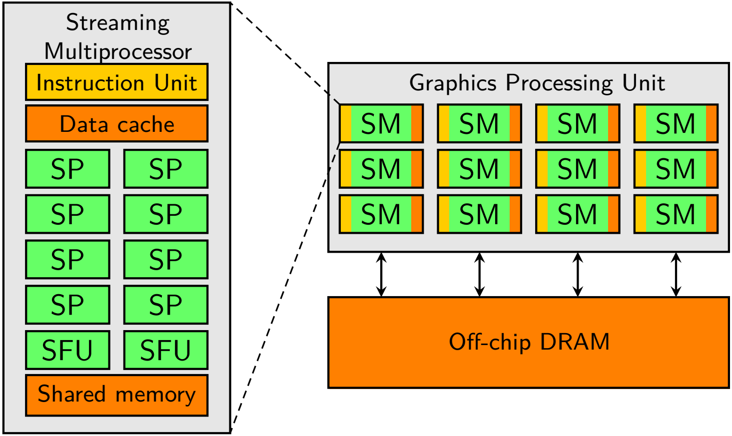
What is CUDA?

- Compute Unified Device Architecture
- Developed by NVIDIA as the architecture for their GPUs
- Scalable architecture suitable for data-parallel tasks
- First CUDA-enabled GPUs released in late 2006
- Programmable
- General-Purpose computing on Graphics Processing Units (GPGPU)

Why use GPUs?

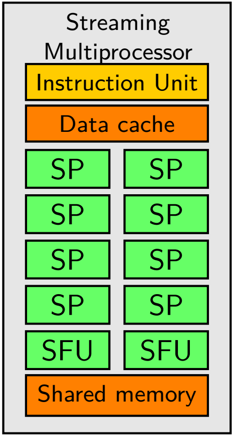

Intel CPUs
NVIDIA GPUs
GTX 295
GTX 280
8800 GTX
8800 Ultra
8800 GTS (G92)
E6600
Q6600
QX6800
QX9775
i7 965
Peak GFLOP/s
1,500
1,000
500
0
Feb 2007
Aug 2007
Mar 2008
Sep 2008
Jan 2009
Date

CUDA hardware architecture



Memory

- Registers
- Shared
- Global
- Local
- Some read-only data caches

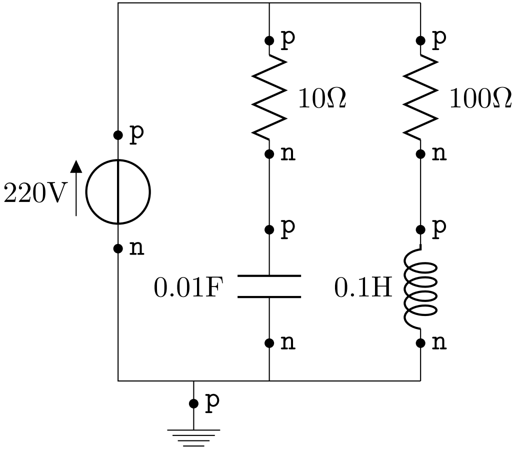


C for CUDA

- Extension of C
- CUDA runtime API
- Host and Device
- Kernels
- Kernels defined with `__global__`
- Kernels launched with `<<<...>>>` syntax
- One kernel executed by multiple threads, divided into blocks.
- Blocks automatically allocated on multiprocessors.

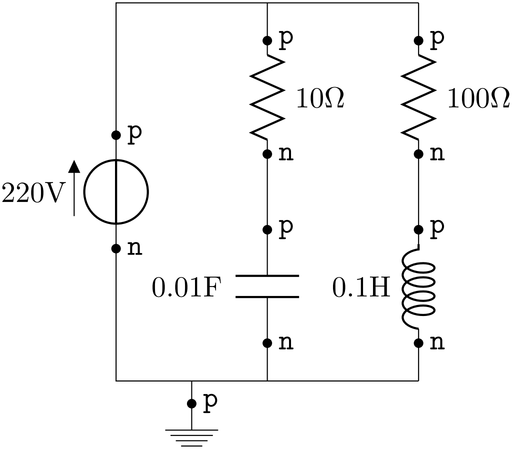
```
// The kernel
__global__ void brighten_pixel(int image[256][256])
{
int pixel_x = blockIdx.x * blockDim.x + threadIdx.x;
int pixel_y = blockIdx.y * blockDim.y + threadIdx.y;

image[pixel_x][pixel_y] += 10;
}

int main()
{
// Invoking the kernel
dim3 grid_dim(16, 16); // 16x16 blocks in the grid
dim3 block_dim(16, 16); // 16x16 threads in each block

brighten_pixel<<<grid_dim, block_dim>>>(image);
}
```

Problem to solve



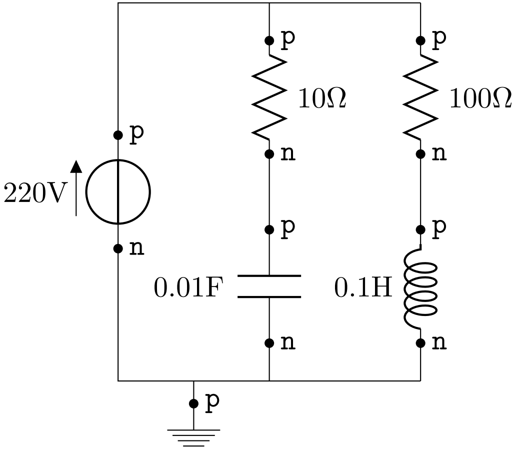
$$u = A \cdot sin(\omega t)$$
$$u_3 = R_2 \cdot i_2$$
$$u_1 = u - u_2$$
$$u_4 = u - u_3$$
$$i_1 = \frac{u_1}{R_1}$$
$$i = i_1 + i_2$$
$$\dot{u}_2 = \frac{i_1}{C}$$
$$\dot{i}_2 = \frac{u_4}{L}$$

Problem to solve



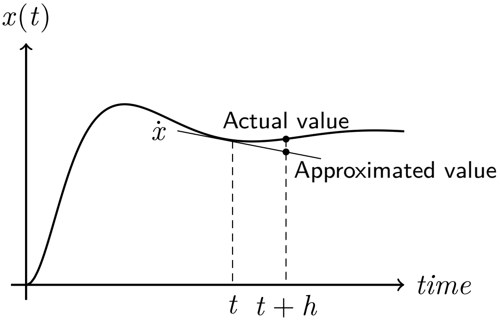
$$u = A \cdot sin(\omega t)$$
$$u_3 = R_2 \cdot i_2$$
$$u_1 = u - u_2$$
$$u_4 = u - u_3$$
$$i_1 = \frac{u_1}{R_1}$$
$$i = i_1 + i_2$$
$$\dot{u}_2 = \frac{i_1}{C}$$
$$\dot{i}_2 = \frac{u_4}{L}$$

Problem to solve



$$u = A \cdot sin(\omega t)$$
$$u_3 = R_2 \cdot i_2$$
$$u_1 = u - u_2$$
$$u_4 = u - u_3$$
$$i_1 = \frac{u_1}{R_1}$$
$$i = i_1 + i_2$$
$$\dot{u}_2 = \frac{i_1}{C}$$
$$\dot{i}_2 = \frac{u_4}{L}$$

Numerical integration



Fourth order Runge-Kutta method (RK4)

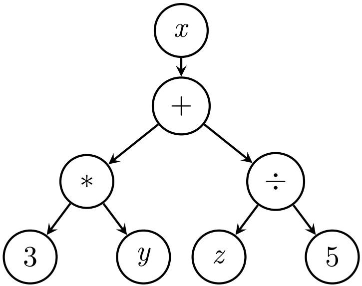
$$\boldsymbol{k_1} = \boldsymbol{f}\left(\boldsymbol{x}\left(t\right), \boldsymbol{u}\left(t\right), t\right)$$
$$\boldsymbol{k_2} = \boldsymbol{f}\left(\boldsymbol{x}\left(t + \frac{h}{2} \cdot \boldsymbol{k_1}\right), \boldsymbol{u}\left(t + \frac{h}{2}\right), t + \frac{h}{2}\right)$$
$$\boldsymbol{k_3} = \boldsymbol{f}\left(\boldsymbol{x}\left(t + \frac{h}{2} \cdot \boldsymbol{k_2}\right), \boldsymbol{u}\left(t + \frac{h}{2}\right), t + \frac{h}{2}\right)$$
$$\boldsymbol{k_4} = \boldsymbol{f}\left(\boldsymbol{x}\left(t + h \cdot \boldsymbol{k_3}\right), \boldsymbol{u}\left(t + h\right), t + h\right)$$
$$\boldsymbol{x}(t+h) \approx \boldsymbol{x}(t) + h \cdot \frac{1}{6} \cdot \left(\boldsymbol{k_1} + 2 \cdot \boldsymbol{k_2} + 2 \cdot \boldsymbol{k_3} + \boldsymbol{k_4}\right)$$

Previous work

- Peter Aronsson: *Automatic Parallelization of Equation-Based Simulation Programs*
- Håkan Lundvall: *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages*
- ModPar

Task graphs

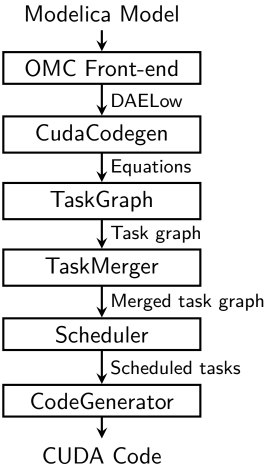
$$x = 3 * y + \frac{z}{5}$$



Task merging

- Aronsson's Task Merging Method (ATMM)
- Reduce the number of nodes that need to be scheduled
- Finding parallelism
- Simpler set of rules compared with ATMM

Overview


Modelica Model
OMC Front-end
DAELow
CudaCodegen
Equations
TaskGraph
Task graph
TaskMerger
Merged task graph
Scheduler
Scheduled tasks
CodeGenerator
CUDA Code

Modelica to task graph

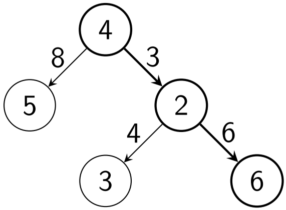
- OMC front-end used to parse models
- CudaCodegen exports equations to external C++ module
- The C++ module builds a task graph
- The task graph is merged and scheduled

Scheduling overview

- Nodes containing tasks scheduled on processors
- Tasks scheduled in correct order

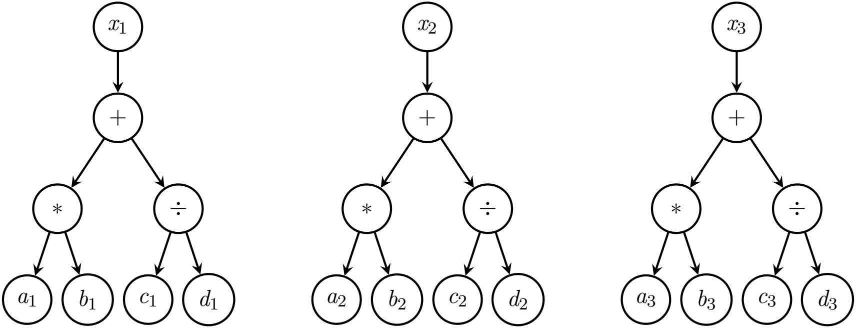
Critical path scheduling

1. Find critical path in task graph
2. Schedule it on processor with least work
3. Remove critical path from task graph
4. Continue until all nodes are scheduled

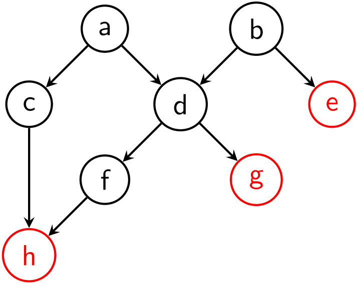


Finding equivalent nodes

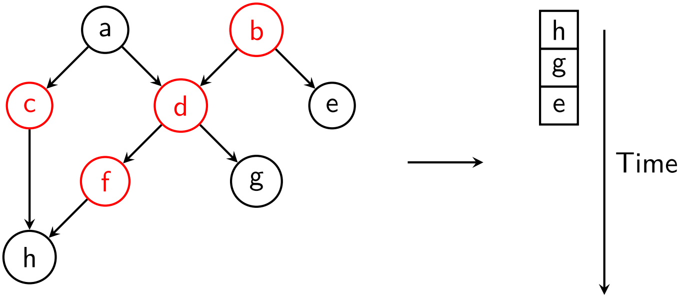
$x_1 = a_1 * b_1 + \frac{c_1}{d_1}$ $x_2 = a_2 * b_2 + \frac{c_2}{d_2}$ $x_3 = a_3 * b_3 + \frac{c_3}{d_3}$



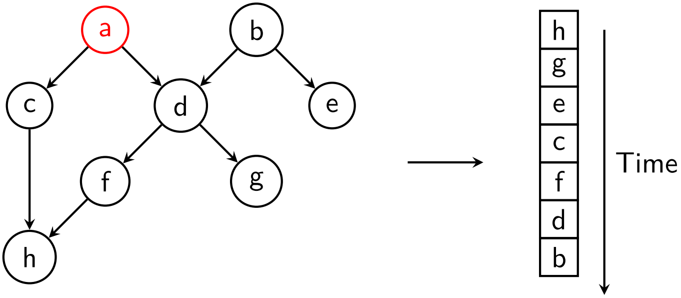
Task scheduling



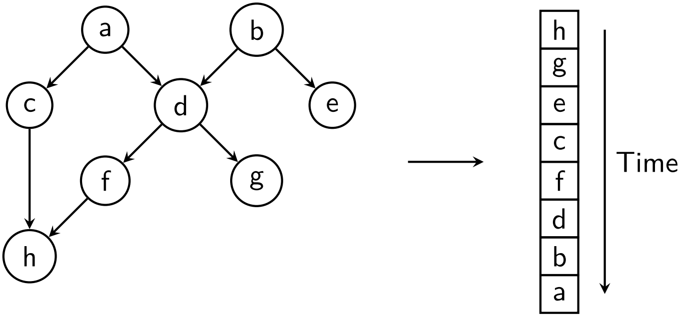
Task scheduling



Task scheduling

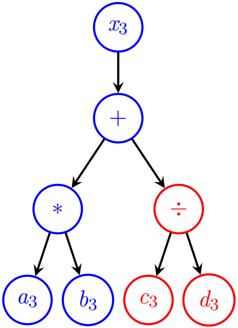


Task scheduling

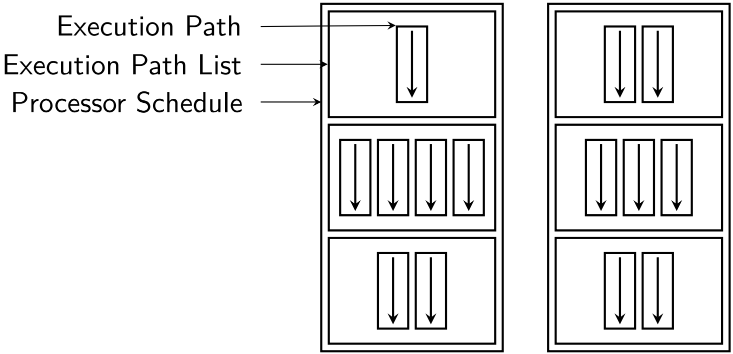


Communication

- Block synchronization not supported by CUDA
- Communicate with global memory
- Locks and signals
- Inefficient
- Limits number of thread blocks



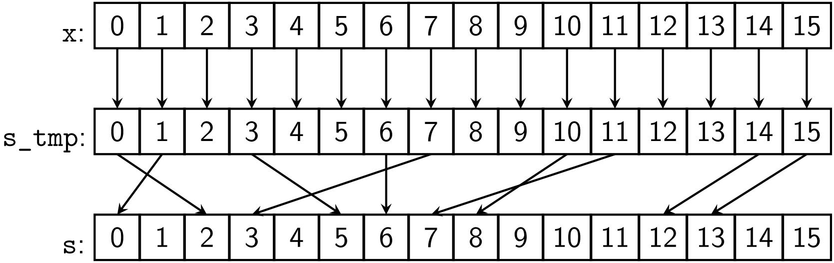
Schedule



Code generation for tasks

- Generate code for tasks based on their type
- Create temporary variables
- Assign results
- Use shared memory for better performance

Shared memory allocation



Task execution example

```
__device__ void execute_tasks_3(real *dx, real *x, real *y, real *c, bool *l, real t)
{
  int id = threadIdx.x;
  extern __shared__ real s[];
  allocate_3(s, dx, x, y);

  if(threadIdx.x < 20)
  {
    real tmp0 = s[1 + 4 * id] + s[0 + 4 * id];
    real tmp1 = -2 * s[2 + 4 * id];
    real tmp2 = tmp1 + tmp0;
    real tmp3 = powf(64, 2);
    s[3 + 4 * id] = tmp3 * tmp2;
  }

  if(threadIdx.x == 0)
  {
    copy_back_3(s, dx, x, y);
  }
}
```

Shared memory allocation example

```
__device__ void allocate_3(real *s, real *dx, real *x, real *y)
{
  real *s_temp = &s[80];
  s_temp[threadIdx.x] = x[2 + threadIdx.x];

  if(threadIdx.x == 0)
  {
    s[0] = s_temp[0];
    s[2] = s_temp[1];
    s[1] = s_temp[2];
    ...
    s[21] = s_temp[17];
    s[24] = s_temp[18];
    s[26] = s_temp[19];
  }

  s_temp[threadIdx.x] = x[22 + threadIdx.x];
  if(threadIdx.x == 0)
  {
   ...
  }

  if(threadIdx.x == 0)
  {
   s[78] = y[1];
  }
}
```

Copy-back function example

```
__device__ void copy_back_3(real *s, real *dx, real *x, real *y)
{
  dx[3] = s[3];
  dx[6] = s[7];
  dx[9] = s[11];
  ...
  dx[57] = s[75];
  dx[60] = s[79];
  y[1] = s[78];
}
```

Spreading the work

```
__global__ void execute_tasks(real *dx, real *x, real *y, real *c, bool *l, real t)
{
  switch(blockIdx.x)
  {
   case 0: execute_tasks_0(dx, x, y, c, l, t); break;
   case 1: execute_tasks_1(dx, x, y, c, l, t); break;
   case 2: execute_tasks_2(dx, x, y, c, l, t); break;
   case 3: execute_tasks_3(dx, x, y, c, l, t); break;
   case 4: execute_tasks_4(dx, x, y, c, l, t); break;
   case 5: execute_tasks_5(dx, x, y, c, l, t); break;
   ...
  }
}
```

Main simulation loop example

```
int shmem_size = 100 * sizeof(real);

for(int step = 0; step < steps; ++step)
{
  r_dx += DERIVATIVES;
  r_x += STATES;
  r_y += ALGEBRAICS;

  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x, d_y, d_c, d_l, t);
  step_and_increment1<<<2, 32>>>(d_x, d_old_x, d_dx, d_k, half_h);

  t += half_h;

  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x, d_y, d_c, d_l, t);
  step_and_increment2<<<2, 32>>>(d_x, d_old_x, d_dx, d_k, half_h);
  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x, d_y, d_c, d_l, t);
  step_and_increment3<<<2, 32>>>(d_x, d_old_x, d_dx, d_k, h);

  t += half_h;

  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x, d_y, d_c, d_l, t);
  step_and_integrate<<<2, 32>>>(d_x, d_old_x, d_dx, d_k, h_div_6);

  cudaMemcpy(r_x, d_x, STATES * sizeof(real), cudaMemcpyDeviceToHost);
  cudaMemcpy(r_dx, d_dx, DERIVATIVES * sizeof(real), cudaMemcpyDeviceToHost);
  cudaMemcpy(r_y, d_y, ALGEBRAICS * sizeof(real), cudaMemcpyDeviceToHost);
}
```

RK4 Solver for OMC

- OMC supports using different solvers
- Only DASSL and Euler implemented so far
- RK4 solver implemented to compare GPU to CPU

Hardware

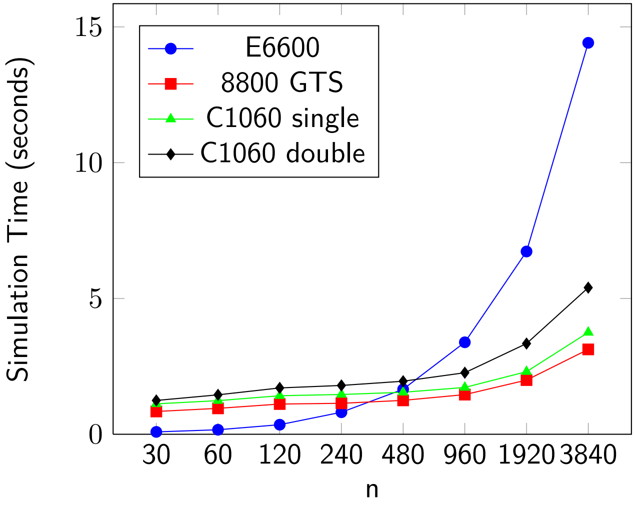
Hardware specifications

	GeForce 8800 GTS	Tesla C1060
Streaming Multiprocessors	12	30
Scalar Processors	96	240
Scalar Processor Clock (MHz)	1200	1300
Single Precision GFLOPS	346	933
Double Precision GFLOPS	N/A	78
Memory Amount (MB)	320	4096
Memory Interface	320-bit	512-bit
Memory Clock (MHz)	800	800
Memory Bandwidth (GB/s)	64	102
PCIe Version	1.0	2.0 (1.0 used)
PCIe Bandwidth (GB/s)	4	8 (4 used)
CUDA Compute Capability	1.0	1.3

Model

```
model WaveEquationSample
  parameter Real L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter Real dL = L/n "Section length";
  parameter Real c = 1;
  Real[n] p(start = fill(0,n));
  Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p);

  for i in 2:n-1 loop
    der(dp[i]) = c^2 * (p[i+1] - 2*p[i] + p[i-1]) / dL^2;
  end for;
end WaveEquationSample;
```

Measurements



Measurement breakdown

	8800 GTS	C1060 single precision	C1060 double precision
Task Execution	0.164	0.592	0.389
Shared Memory	1.440	1.426	2.287
Integration	0.417	0.400	0.445
Memory Transfers	1.104	1.332	2.278

Conclusions

- Possible to get significant speedups by using GPU
- Perhaps a hybrid CPU+GPU approach is better though
- Reducing generated code size necessary for larger models
- New architecture next year: Fermi
- OpenCL