BEAM DYNAMICS SIMULATIONS USING GPUS

J. Fitzek, S. Appel, O. Boine-Frankenheim, GSI, Darmstadt, Germany

Abstract

PATRIC is a particle tracking code used at GSI to study collective effects in the FAIR synchrotrons. Due to the need for calculation-intense simulations, parallel programming methods are being explored to optimize calculation performance.

Presently the tracking part of the code is parallelized using MPI, where each node represents one slice of the particles that travel through the accelerator. In this contribution different strategies will be presented to additionally employ GPUs in PATRIC and exploit their support for data parallelism without major code modifications to the original tracking code. Some consequences of using only single-precision in beam dynamics simulations will be discussed.

PATRIC SIMULATION CODE

The international FAIR facility with its new accelerators will be built at GSI, using the existing linac and SIS18 synchrotron as injectors. PATRIC is a particle tracking code that has been developed at the GSI accelerator physics department over many years and that is used to study collective effects in the circular accelerators within the FAIR facility. For more information on PATRIC see also [1].

Structure of the Code

Besides others, the PATRIC simulation code mainly consists of the Pic class as representation of the particles, the SectorMap class for ion optical elements like magnets including their transfer matrix, and the BeamLine class to group SectorMaps to form the accelerator (see also Figure 1). The main program takes care of the object creation, distribution of the calculation, and time measurement.

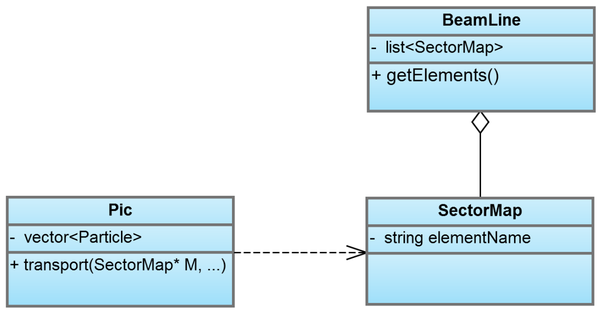

Figure 1: Structure of the PATRIC simulation code.

Existing Problem Division

Presently the tracking part of the code is parallelized using MPI, where each node represents one slice of the particles that travel through the accelerator. For tracking many thousand particles, a large number of identical calculations has to be performed on different data. Therefore the decision was to divide the problem by the data.

Dividing the data can be done in many different ways. As basis for PATRIC, the particles where diveded longitudinally in slices and assigned to MPI nodes as shown in Figure 2. Because particles that interact with each other reside on the same node, this distribution allows to include the calculation of these effects locally on the MPI node. By dividing the problem as described, PATRIC is enabled for distributed computing, which already gives a good performance gain.

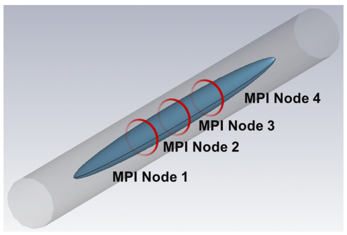

Figure 2: Structure of the existing parallelization using MPI: Each node processes one longitudinal slice of particles (figure created with CST EM Studio).

Due to the need for even more calculation-intense simulations for FAIR, parallel programming methods are being explored to optimize the calculation performance of the PATRIC simulation code beyond the possibilities of MPI. In the context of a diploma thesis different strategies are being investigated to additionally employ GPUs in PATRIC and exploit their support for data parallelism without major code modifications to the original tracking code.

Parallelization with GPUs

Today all end user computers contain powerful GPUs with remarkable floating point performance. GPUs are specialized on graphics processing and therefore support massively parallel calculations. Since they are affordable and nowadays equiped with a general purpose programming interface, they are more and more used to solve calculation-intense problems. Especially their support for data parallelism makes them suit for the calculations that have to be performed during particle simulations.

ISBN 978-3-95450-116-8

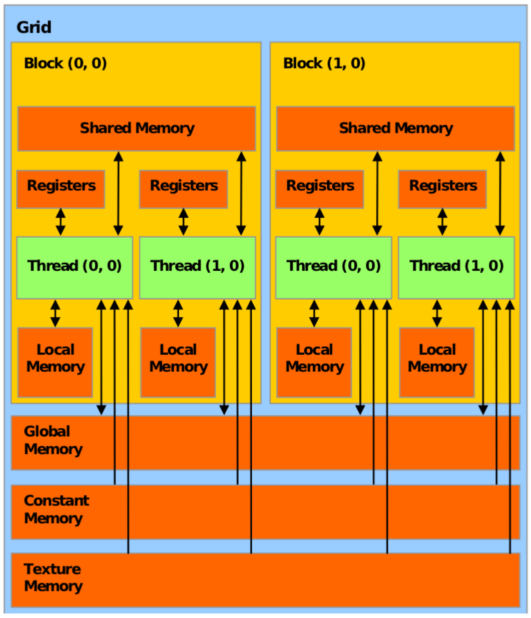

Figure 3: CUDA device memory model, NVIDIA, [2].

Mapping of the Problem

For the task to include GPUs in the existing code, an NVIDIA GPU was used together with CUDA programming. The CUDA device memory model as shown in Figure 3 allows to achieve a good structuring of the code by mapping the problem onto threads, blocks and grids. Since blocks are executed independently from each other, scalability is ensured, and the code can seamlessly be executed on e.g. a more powerful GPU without the need to rewrite it. The drawback of GPU programming is the fact that to copy data from the host to the device memory and back is very time consuming. Therefore the challenge is to find a good problem distribution onto the CPU and GPU to gain from the calculation performance benefit of the GPU.

Development of GPU Code

To develop code for NVIDIA GPUs, different programming models exist that allow either to program on a low level using CUDA for C++ [2] or to program on a high level abstraction using the template library Thrust for CUDA [3]. These possibilities are shortly presented using the example of the transfer function, i. e. a matrix-vector-multiplication as shown below.

```
for(int j=0;j<6;j++) {
   R1[j]=0.0;
   for(int l=0;l<6;l++)
     R1[j]+=T[j*6+l]*R0[l];
}
```

Figure 4: Code snippet: Original multiplication

Using the low level programming model CUDA for C++ has the advantage, that all features of the NVIDIA GPUs are available to the programmer. The API gives full control over the GPU and therefore allows to assign calculation code directly to parallel execution units by using CUDA threads, as shown in the example in Figure 5. By structuring the execution using threads, blocks and grids, the problem distribution can be handled individually. The drawback of this solution are many lines of technical code that “pollute” the original algorithm code, which leads to more complex code and reduced maintainability.

```
cudaMalloc(..);
cudaMemcpy(..);
kernel<<<dimGrid, dimBlock>>>(d_T,
  d_particleVectors, numberOfParticles);

__global__ void kernel(...) {
  (...)
  for (int i = 0; i < 6; i++) {
    value += transportMatrix[threadIdx.y * 6 + i]
      * particleVectors[particleVecStart + i];
  }
  __syncthreads();
  particleVectors[elementToCalculate] = value;
  __syncthreads();
}
```

Figure 5: Code snippet: Multiplication using CUDA for C++, each parallel execution unit calculates one element of the result vector.

Using the high level programming model Thrust for CUDA has the advantage, that an abstraction of the GPU is presented to the developer. Thrust supports differnet GPUs and programming models and is therefore used whenever interoperability is needed. Since Thrust provides many predefined functions, it also has a quick learning curve and supports rapid prototyping. The code stays clear of too many technical codelines which leads to less complex code and better maintainability. Drawback of using Thrust is that it does not provide full control over the GPU, for special purposes the low level programming must anyway be used. However, it is possible to mix these two programming models. Another aspect is that some predefined functions only support single precision floating point operations, though the underlying GPU is capable of double precision (see next section).

```
thrust::device_vector<float> x_device_vector;
x_device_vector = x_host_vector; //copy

void transport_gpu(Tmat& T, Pic_gpu& pics){
  (..)
  thrust::transform(pic_first, pic_last,
    pic_first, rotate_pic(T));
}
```

Figure 6: Code snippet: Example for using Thrust for the particle transport.

FLOATING POINT CALCULATION

Besides the challenges of code structuring, another important aspect is the fact, that older GPU models only support single precision floating point operations, which suffices for standard graphics processing but can be a problem for calculation-intense physic problems with chaotic behavior. In the past, lots of problems were not ported to the GPU because of this issue. Today, most of the recent GPU models contain double precision support with lower performance, but the performance continually gets better (up to 80% with the latest GPU models). However, not all APIs include double precision support yet. In the context of this work, this fact must also be taken into account.

FIRST APPROACHES TO EMPLOY GPUS

To employ GPUs in the PATRIC simulation code, a first approach is to simply perform the transport routine on the GPU (GPU-Version 1 in Figure 7). The static transfer matrices reside in the global memory of the device (the GPU). During each transport step, the particle vectors are copied to the device, the transfer is calculated and the result is copied back to the host. This approach is simple but as measurements show the execution time rises due to the copy overhead in each step.

A second approach is to keep also the particles in the global memory of the GPU (GPU-Version 2 in Figure 7). Transport steps can then be performed only on the GPU which eliminates most of the copy steps. However, intermediate calculations at predefined steps during the overall calculation must then either be done also on the GPU or – as one requirement is to keep existing calculation routines – intermediate particle information must be copied back to the host to perform these calculations.

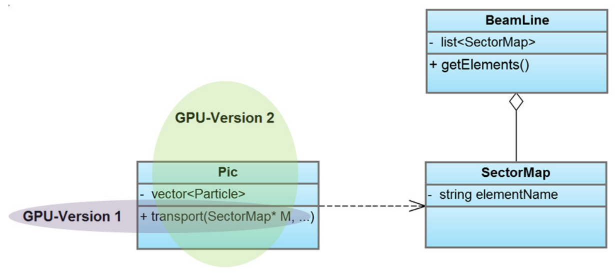

Figure 7: PATRIC code with GPU-enabled parts.

FIRST MEASUREMENTS

For first measurements, a desktop computer with six Intel 2.67GHz CPUs and one NVIDIA Tesla C2075 GPU was used. The code was executed with a constant focussing optic, 10000 macro particles, and 128 cells. The number of MPI processes was set to one, to exclude impacts on the performance due to oversubscription of the CPUs and due to the fact, that only one single GPU was available. Average execution times were taken from 20 runs. Figure 8 shows the results of the first measurements. The 10 seconds longer execution time of the GPU-Version 1 compared to the original version can be explained with the copy overhead within each transport step. This approach seems not ideal, since too many copy steps outweigh the calculation performance gain. This leads to the conclusion, that larger parts of the calculation must be moved to the GPU to achieve a performance gain. As a next step, the GPU-Version 2 approach will be implemented and measured; positive speedup is expected when the particles can be always kept on the GPU.

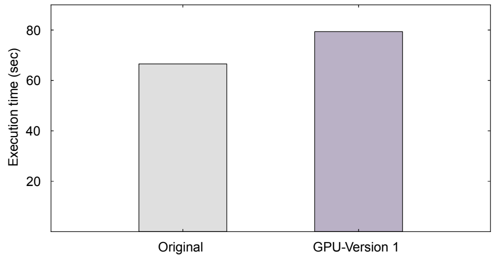

Figure 8: First measurements of the code variants.

OUTLOOK

Employing GPUs in the PATRIC code will be elaborated in the context of a diploma thesis for the GSI accelerator physics department, supervised by Prof. Dr. Keller, subject Parallelism and VLSI at the FernUniversität Hagen. As part of this diploma thesis, different variants will be implemented and measured, including different structuring of the code, usage of the low or high level API (performance versus usability), and the utilization of single or double precision (performance versus error cumulation).

Already the first tests indicate, that the copy overhead is one of the limiting factors. Structuring the code in such a way, that bigger amounts of related calculations can be performed on the GPU might be a promising approach. Therefore, the idea to completely keep the particles on the GPU will be investigated further. The constraint is that it must still be possible to include existing particle quantity calculations (e. g. emittance) at predefined points in the overall execution.

If employing of GPUs succeeds, future extensions are foreseen that include e.g. calculation of space charge effects and particle kicks locally on each node again by using the GPU do do these calculations.

REFERENCES

[1] O. Boine-Frankenheim, V. Kornilov, "Simulation of Transverse Coherent Effects in Intense Ion Bunches", this conference, TUAAI3.

[2] NVIDIA Corporation, "CUDA Programming Guide", Santa Clara USA, 2007.

[3] Thrust, open source parallel algorithms library, available at `http://code.google.com/p/thrust`

ISBN 978-3-95450-116-8