The magic determination of the magic constants by ttgLib autotuner

Mikhail Pritula[1,2], Maxim Krivov[1,2], Sergey Grizan[2], Pavel Ivanov[1,2]
{ M_Pritula, M_Krivov, S_Grizan, P_Ivanov }@ttgLabs.com

[1] M.V. Lomonosov Moscow State University (Russia), [2] ttgLabs, LLC

Moscow State University

MSU CUDA Center of Excellence*

ttgLabs, LLC

Concept

Target problems

- During the optimization of GPU applications a lot of magic constants are often introduced (such as block size, grid decomposition, etc). The optimal values for these constants depend on the hardware and input data being used. Therefore it's really difficult to define optimal values that will provide the best performance in all usage scenarios.

- Support of multi-GPU and CPU+GPUs systems requires good load balancing strategy. E.g. for small data only CPU should be used, for bigger one - one GPU, for much more bigger – all GPUs. And again it depends on processing data as well on used hardware.

The solution

The autotuning concept that was implemented in ttgLib library allows to solve all these problems at runtime. The presented library gathers statistics about the program being run and dynamically adjusts magic constants for the best performance and provides smart load balancing.

Load balancing use cases

- Ignoring some computing devices for better speed-up
- Taking into account GPU performance volatility
- Considering non-linear GPU Performance dependency from data size
- Minimizing side effects from non-computing CPU threads / processes

Magic constant determination use cases

- Selecting the best grid decomposition
- Detecting optimal «weight» for GPU threads
- Choosing the best buffer sizes for asynchronous operations
- Testing different unrolling for loops
- Switching between different branches

Code samples

Sample 1. Detecting the optimal grid

```
Parameter<int> blockWidth = 32;
//After each launch of this code real value
//of blockWidth will be changed in order to
//achieve better performance
dim3 threads(blockWidth);
dim3 grid(size + 1 / blockWidth);
someCudaKernel<<<grid, threads>>>();
```

Sample 2. Selecting the best branch

```
void cudaReduce_SharedMem(void *data)
{/*Performing reduction using shared memory*/}

void cudaReduce_Atomic(void *data)
{/*Performing reduction using atomic operations*/}

HybridTask hTask;
hTask.CUDA() += cudaReduce_SharedMem;
hTask.CUDA() += cudaReduce_Atomic;
//After N-th call the best branch will be selected.
hTask.Execute(someData);
```

Sample 3. Detecting the best "weight" of GPU-thread

```
__global__ void SomeCudaKernel(int pointsPerThread)
{
   for (int i = 0; i < pointsPerThread; i++)
      { /*Computing*/ }
}

Parameter<unsigned int> pointsPerThread = 8;
//Optimal value will be selected for good SM utilization.
SomeCudaKernel<<<grid, threads>>>(pointsPerThread);
```

Sample 4. Performing load balancing

```
void cudaKernel(void *data, size_t lo, size_t hi)
{/*Performing computations using GPU and shared memory*/}

void sseKernel(void *data, size_t lo, size_t hi)
{/*Performing computations using CPU and SSE*/}

HybridFor hFor;
hFor.CUDA() += cudaKernel;
hFor.Serial() += sseKernel;
//Array will be processed on all available devices. Or
//using only some of them – it depends on runtime specific.
hFor.Process(someData, someData_size);
```

Tuning algorithms

Background

All magic constants (declared via Parameter<> template or hybrid programming primitives) is being mapped into hypercube. After that autotuning process can be considered as minimization of time functional defined in the constructed hypercube. Here coordinates correspond to constant's values, and the time function measures iteration execution time.

As a result autotuning can be enabled by performing the following steps:

Spep 1. Making constants dynamic by declaring them via Parameter<> template class.

Step 2. Annotating iterations that correspond to time function measuring

For more information visit ***http://ttgLabs.com***

Used algorithms

Currently, the following optimization algorithms were developed:

- **Clicking.** Splits hypercube into chunks with weights, after that takes the best one and splits it into new chunk set and so on. Optimal for tuning when small amount of iterations available.

- **Genetic.** Creates and manages the population of values. Most suitable for a big amount of optimizing constants and a lot of available iterations.

- **Sticky.** Recognizes hardware and performs explicit load-balancing by trying to ignore the worst devices and evenly distribute load. All other constants will be optimized by clicking or genetic algorithm.

- **LUT.** Builds LookUp table for each computing device that maps data size into expected performance. Uses constructed table for load balancing while other constants will be optimized by clicking or genetic algorithm.

Results (one node)

Laplace equation

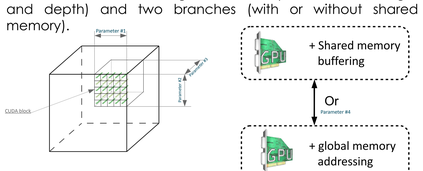
Problem statement

Solution of three dimensional Laplace equation with Dirichlet boundary condition on structured grid using Jacobi method.

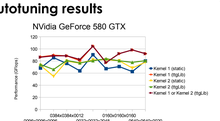
Implementation details

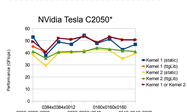
Whole area is being split into a set of chunks. The width and the height of each correspond to 2D CUDA block that is being processing on one SM, the depth is the count of points being processed on one thread.

This test uses two computing CUDA kernels – (1) with global memory addressing and (2) width shared memory buffering. Here we have three magic constants (chunk width, height and depth) and two branches (with or without shared memory).



Autotuning results





Heat-Mass transfer equation

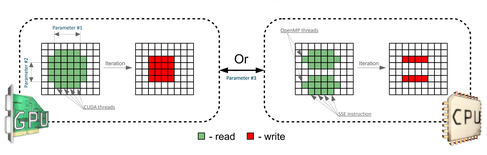
Problem statement

Solution of two dimensional parabolic equation with Neumann boundary condition on structured grid using explicit 6-points scheme

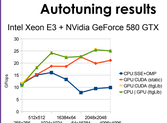
Implementation details

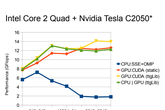
This test consists of two computing kernels – (1) version for CPU (SSE and OpenMP) and (2) version for GPU (CUDA). In the GPU kernel the width and the height of CUDA blocks are defined as optimization parameters. For small data CPU is preferred while GPU provides better performance for bigger grids.

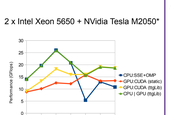
As a result this test provides two computing kernels (branches), and each of them contains two magic constants.



Autotuning results







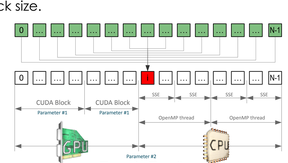
N-Body problem

Problem statement

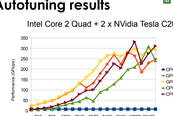
System of N bodies each of which affects others is being simulated in three dimensional space. Euler method is used for numerical integration of 2*N differential equations.

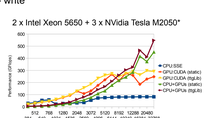
Implementation details

As this problem has $O(N^2)$ complexity three computing kernels were created – (1) version for CPU that uses SSE and OpenMP, (2) version for GPU that uses CUDA and performs simulation without synchronization with CPU and (3) version for CPU+GPUs that uses all available computing devices and performs data synchronization after each iteration. The second and the third kernels provide one magic constant – CUDA block size.



Autotuning results





Results (cluster)

Euler equations

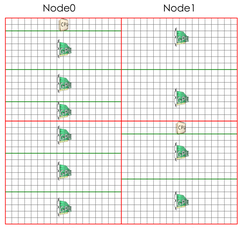
Problem statement

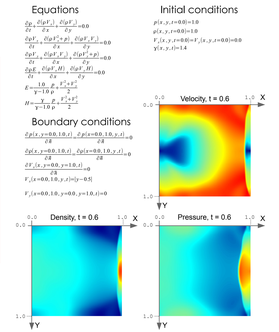
System of the two dimensional Euler equations with mixed boundary condition is being solved on structured grid using Roe solver.

Implementation details

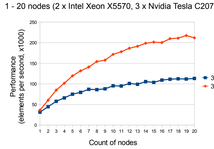
Each node has two CPUs and three GPUs. Therefore overall simulation process was split into two types of iterations – (1) global iterations across all nodes and (2) local iterations across computing devices of a single node.

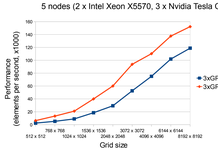
In global iterations one magic constant was introduced that defines the depth of neighbor areas overlapping in order to minimize MPI synchronization rate. In local iterations HybridFor primitive is being used for load balancing between CPUs and GPUs. Finally, each CUDA-kernel uses two magic constants for CUDA-block determination.

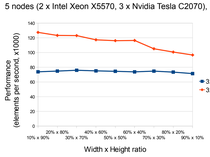




Autotuning results







* Results that are presented in this work were achieved using supercomputing resources of Lomonosov Moscow State University