Large-Scale Loops Parallelization for GPU Accelerators

Anatoliy Doroshenko, Oleksii Beketov

Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
doroshenkoanatoliy2@gmail.com, beketov@isofts.kiev.ua

Abstract. The technique that allows to extend GPU capabilities to deal with data volumes that outfit internal GPU's memory capacity is developed. The technique involves loop tiling and data serialization and could be applied to utilize clusters consisting of several GPUs. Applicability criterion is specified. Transforming scheme designed and semiautomatic proof-of-concept software tool are implemented. Conducted an experiment to demonstrate the feasibility of the proposed approach.

Keywords. Parallelization methods, loop optimization, GPGPU, CUDA.

1 Introduction

Loop parallelization is a long-standing problem of computational programming. Loops give a fair parallelization opportunity for numerous scientific modeling problems that involve numerical methods. Along with spreading of GPGPU technology [1] that allows employment of graphics accelerators for solving computational tasks new challenges arises. As far as GPU is not a standalone device and is managed by a host operating unit, it should be considered within the context of heterogeneous computational platforms. Composing the programs for such the platforms demands knowledge in architecture and specific programming tools. Generally, the concurrent software development passes through the stage of successive implementation that becomes a starting point for further platform-dependent and hardware environment specific implementations.

The existing automatic code parallelizing tools [2, 3, 4] don't account the limited amount of GPU's on-board memory space while real-life problems demand in huge amounts of data to be processed. To embrace those cases of massive computational tasks that involve large amounts of data we propose a technique that provides an ability to rip the loop and to split the data and calculation operations.

2 Loop Transformations

This section is introducing an idea of loop rearrangement. Let's consider a nested loop that consists of *for* type operators of the following form:

```
$for \quad 0 \le i_N < I_N :$
  $for \quad 0 \le i_{N-1} < I_{N-1} :$
    ...                                                        (1)
      $for \quad 0 \le i_0 < I_0 :$
        $F(\vec{i}, D),$
```

where $I_k \in \mathrm{N}$, $0 \le k \le N$, $for \quad 0 \le i < I : S(i)$ is a notion for a sequence of $\{S(0), S(1), ..., S(I)\}$, $F : I_0 \times ... \times I_N \times D \mapsto D$ is a mapping over the dataset D. We'll call i_k a *k*-th counter of the cycle (1), $\vec{i} = (i_0, i_1, ..., i_N) \in I_0 \times ... \times I_N$ a vector of counters and a particular call of $F(\vec{i}, \cdot)$ for some specified value of $\vec{i}$ an iteration. First, let's make the following substitution for each of the *for* statements:

$$for \quad 0 \le i_n < I_n \quad \mapsto \quad \begin{array}{l} for \quad 0 \le s_n < S_n : \\ \quad for \quad s_n \cdot L(I_n, S_n) \le i_n < \min\left((s_n + 1) \cdot L(I_n, S_n), S_n\right), \end{array}$$

where

$$L(a, b) = \left\lfloor \frac{a}{b} \right\rfloor + 1 - \delta_{0, a \bmod b},$$

$\lfloor \cdot / \cdot \rfloor$ denotes quotient, δ is a Kronecker delta and S_n – the desired number of sections for each of the loops to be subdivided, $1 \le S_n \le I_n$. This transformation is commonly known as loop tiling and is performed in optimizing compilers to modify memory access patterns for improving cache access [5]. After substitution and reordering the new loop takes the form

```
$for \quad 0 \le s_N < S_N :$
  ...
    $for \quad 0 \le s_0 < S_0 :$
      $for \quad s_N \cdot L(I_N, S_N) \le i_N < \min((s_N + 1) \cdot L(I_N, S_N), S_N):$      (2)
        ...
          $for \quad s_0 \cdot L(I_0, S_0) \le i_0 < \min((s_0 + 1) \cdot L(I_0, S_0), S_0):$
            $F(\vec{i}, D).$
```

The inner loop is similar to the initial loop but of a diminished scale. Leaving the inner *N+1* loops, let's group the first *N+1*:

```
$for \quad 0 \le e < \prod_{i=0}^{N} S_i :$                    (3)
  $\vec{i} = g(e);$
```

Here $g(\cdot)$ is a mapping that restores the vector of counters $\vec{i}$ and is constructed the following way:

$$g_0(e) = e \bmod S_0,$$
$$g_k(e) = \left\lfloor \left(e - \sum_{j=k+1}^{N} g_j(e) \prod_{l=0}^{j-1} S_l \right) \Big/ \prod_{j=0}^{k-1} S_j \right\rfloor, \ 0 < k < N,$$
$$g_N(e) = \left\lfloor e \Big/ \prod_{j=0}^{N-1} S_j \right\rfloor,$$
$$0 \le e \le \prod_{k=0}^{N} S_k .$$

Loop (3) maintains the sequence of vector of counters equal to the sequence produced by the initial loop (1).

Let's denote the inner *N+1* loops of the cycle (2) along with $g(e)$ as a $kernel(e)$. We intend to delegate the *kernel* execution to GPU and to run it concurrently thus diminishing the depth of the inner loop nest. As the GPU's memory space is isolated from the host's device one, we introduce *serialize* operation that is to prepare the input data required to perform calculations for the step *e* and *deserialize* operation to store the output data processed by GPU. The further implementation of these procedures is out of our scope and depends on the particular problem. Finally, we got:

```
for  $0 \le e < \prod_{i=0}^{N} S_i$ :
  serialize(e, inputData, dataPull);
  transfer2device(inputData);                         (4)
  kernel(e, inputData, outputData);
  transfer2host(outputData);
  deserialize(e, outputData, dataPull);
```

Iterations of the loop (4) could be distributed over concurrently running threads through involving several additional data exchange buffers. This approach could be applied to any distributed memory computational system, e.g. GPU cluster or heterogeneous cluster of any other computation empowered devices. To preserve equivalence in a sense of output results equality for the same given input data Bernstein's [6] conditions must be met. This roughly means that iterations should not overwrite the other's iterations input data and should store their output data apart. The set of S_k, $0 \le k \le N$ are transformation's tunable parameters that are chosen in a way to satisfy Bernstein's conditions and to optimize processing time that is to find a trade-off on time spent on data preparation, transfer and kernel execution. These timings depend on the input and output data load size which is restricted by the total available amount of GPU's memory and the hardware configuration parameters such are input and output memory transfer rate and GPU compute capabilities.

3 Program Execution Flow

Let's consider the node that consists of one multicore CPU and one GPU. Modern GPUs support direct memory access technology thus allowing to proceed data transfer and kernel execution asynchronously. To optimize data exchange process dual buffering is involved. Four buffers at both host and device sides are involved – two for the input and two for the output data exchange. On the host side, calculations proceed in two threads that execute kernel, serialize and deserialize procedures simultaneously. One of them serialize input data and fills the input data buffer, then transmits the buffer to the GPU and launches the kernel, and the second receives output data buffer from GPU and deserialize it. Besides the calculations, GPU carries bidirectional data

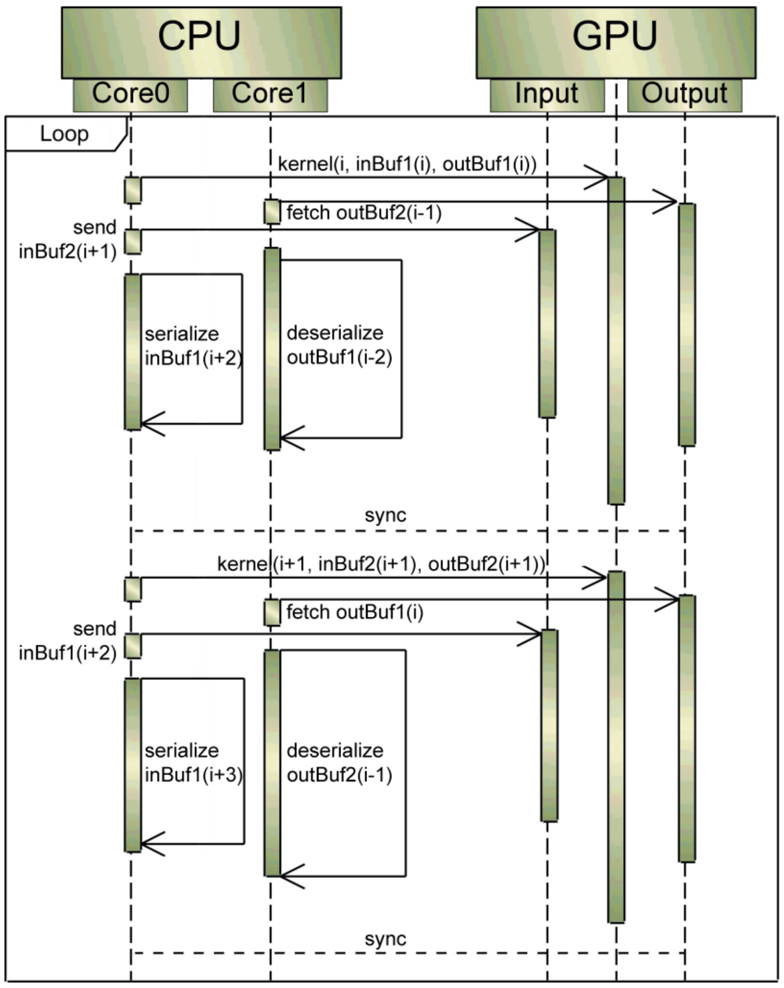

Fig.1. Execution flow diagram of the cyclic stage of the concurrent program for the system of one accelerator and two control flow threads with four data exchange buffers

transfers through the asynchronous data transfer mechanism. Calculations are performed in three stages – initial, cyclic and finalizing.

At the starting point, data buffers are empty, and GPU awaits the data transfer. It doesn't matter what of the threads will carry the initial step as all of the operations are run successively and asynchronous transfer mode is not involved. At the initial step, CPU serializes input data buffers of the first two iterations and transfers the buffer containing the first iteration data to the accelerator.

After the initial step, the cyclic stage starts. The execution flow is shown at the diagram at Fig.1. On the diagram, the iteration's number of which the data is stored in the buffer is given after the buffer's name. One step of the cyclic lap divides into odd and even parts. Both odd and even parts of the first step skip deserialization as the host output buffers are empty yet. At the odd part of the first step, the accelerator-to-host transfer is omitted too. Meanwhile, an accelerator performs calculations over the current buffer, host threads fetch data buffer from the previous step, deserialize penultimate step buffer, send input data buffer for the next step and serialize buffer for the after the next step. In one step two kernel launches are executed. After each of the parts, odd or even, is finished the processes synchronize. Two final steps depend on the actual kernel launches number. If the number of kernel launches is odd, the final step of the cyclic part excludes an even part and does not involve serialization and host-to-accelerator transfer, and the even part of the penultimate step skips serialization. Otherwise, if the number of launches is even the last cyclic loop step is full, but the even part of the final step omits serialization.

The finalizing step deserializes output data buffer transferred at the last cyclic loop step and then fetches and deserializes the final output data buffer consequently finishing the calculations.

4 Application of the Proposed Approach for Constructing a CUDA Program

In this section, we illustrate the application of the proposed approach to matrix multiplication and N-body problems. The time measurements were collected on the hardware system composed of Intel Core i5-3570 CPU (4 cores 3.8Hz) with 16Gb of host memory and NVIDIA Tesla M2050 GPU (3Gb global memory, 384 bits memory bandwidth, connected through PCIe2.0 x8) running Ubuntu 16.04 host OS.

A semi-automatic source-to-source code transformation tool based on the TermWare rewriting system [7] aiming to assist in constructing a new concurrent program was implemented. It takes the initial loop marked with pragmas, applies the transformations (3) and provides with a template of the code of a new loop to be substituted. The remaining actions include serialization and deserialization routines implementation; the kernel could be implemented as well as generated by another tool and adapted in place.

The algorithm of the initial sequential matrix multiplication program involved three-dimensional loop nest. It was transformed using the proposed technique and C-to-CUDA compilers PPCG and Par4All. Both of the programs generated by PPCG and

P4A showed comparable results. After applying the slicing technique the initial matrices were split into submatrices. The internal loop subdivision parameter S_0 was set to 1, the roles of parameters S_2 and S_1 is adjusting submatrices width. The schema with double data exchange buffering and two CPU threads was used. Even not involving GPU, adjusting the slicing number allowed to reach about 12 times acceleration over the initial loop due to CPU caching. For the GPU implementation, the parameterized PPCG generated kernel was used; the source codes of the constructed matrix multiplication program are available at GitHub [8]. The chart at Fig. 3 shows the constructed program's timings and the timings of the program obtained with PPCG relatively to the matrix dataset size. The relative acceleration of about 430 times in comparison to the sequential program executed on the CPU for the datasets of square matrices of the size of 5000x5000 single precision floating point numbers was reached. It could be seen from the chart that the PPCG generated program has achieved maximum data set size less than 300 Mb that is 10% of GPU's global memory available. This is due to the fact that PPCG is limited to static memory usage, thus blocking to link programs with too large static arrays. It is worth to mention that involving two CPU threads is excessive as the part of serialization and deserialization is negligible compared to GPU kernel computation time, that could be seen from the GPU execution profile given at Fig.2. Thus involving just one thread instead won't decrease performance substantially, however, two concurrent threads are required to avoid gaps in kernel launches and to gain maximum performance from the GPU.

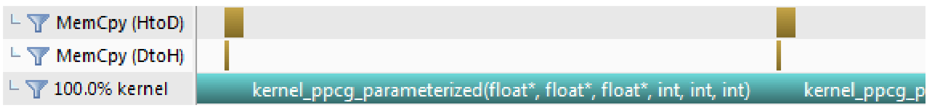
MemCpy (HtoD)
MemCpy (DtoH)
100.0% kernel
kernel_ppcg_parameterized(float*, float*, float*, int, int, int)
kernel_ppcg_p

Fig. 2. The fragment of the profile of matrix multiplication execution.

Another application investigated was a predictor routine from N-body problem with predictor-corrector time-iterative [9] algorithm. The model of the system consists of a set of particles that interact pairwise PPCG applying caused a slowdown effect and led in about 500 times decrease in performance in comparison to the sequential CPU implementation. As for the constructed program with a self-implemented kernel not involving shared memory usage, the relative CPU to GPU acceleration at the selected data size range reached 13 times. The plot at Fig.4 shows the dependency of successive CPU and transformed GPU programs execution time on the data size that is scaled by the alteration the number of particles N. The timings are measured for the one time-step of the prediction routine. The memory limit was not reached as it would take approximately 30 years to process one time-step of the algorithm for a fully loaded GPU that was used in the experiment, however, the applicability of the approach is confirmed.

Thus the difference in the constructed multiplication and N-body programs consists in the kernel, serializer and deserializer implementation while the control flow structure remains identical.

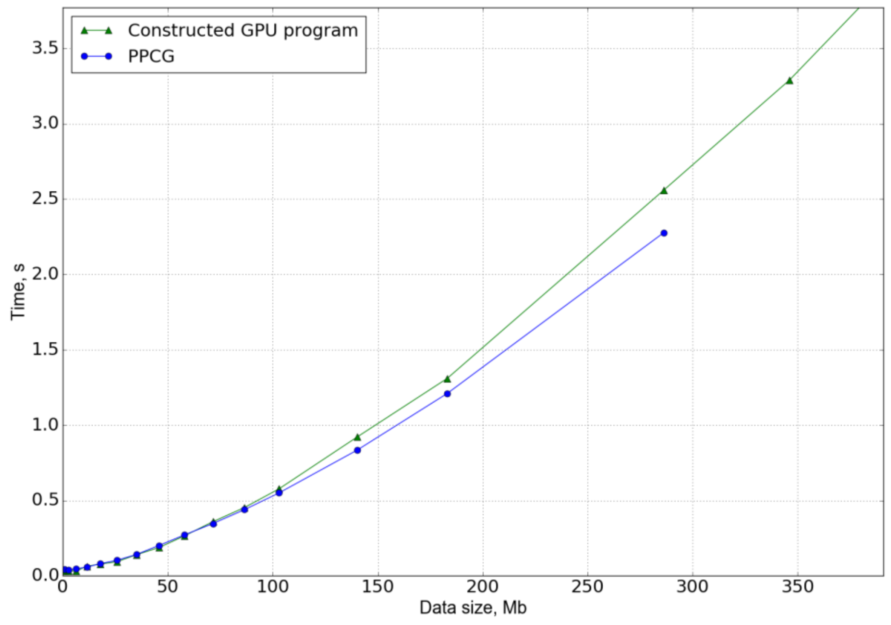

Fig. 3. The dependency of the execution time on the size of the input data for the concurrent PPCG-generated and constructed matrix multiplication programs.

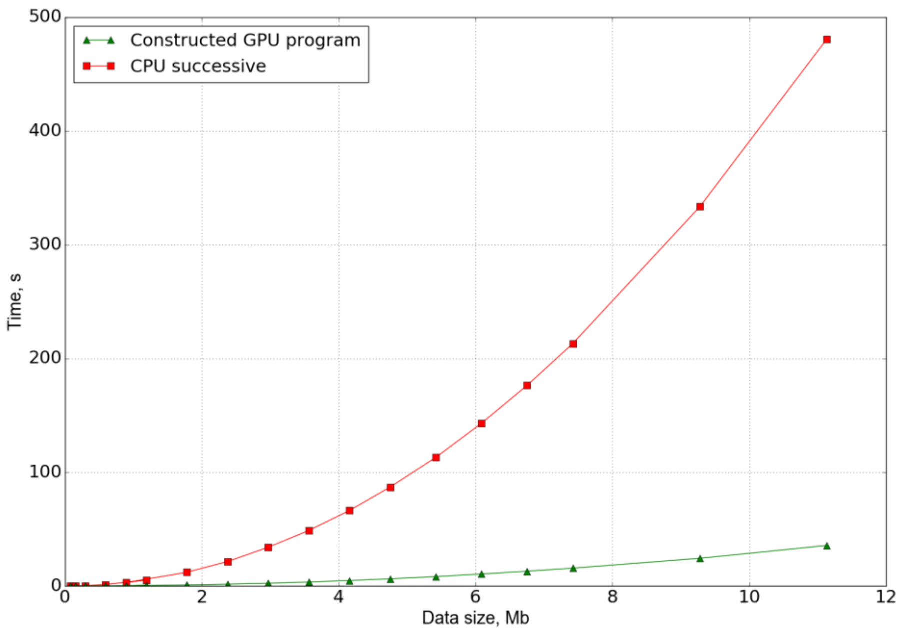

Fig. 4. The dependency of the execution time on the size of the input data for the successive and constructed concurrent N-body programs.

5 Conclusion

This paper proposes an approach for semi-automated parallelization of nested loops for graphics processors. The approach is illustrated by the development of CUDA programs for solving matrix multiplication and N-body problems. As a result, a common unified scheme was used to parallelize both multiplication and N-body problems. An assistant semi-automatic code transformation tool was implemented. Further work relates to developing unified methods and tools in loop parallelization.

References

1. Harris, M. J.: Real-Time Cloud Simulation and Rendering. University of North Carolina Technical Report #TR03-040 (2003).
2. HPC Project, “Par4all automatic parallelization”, http://www.par4all.org
3. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: An overview of the pips project. In ACM Int. Conf. on Supercomputing (ICS’2), Cologne, Germany (June 1991).
4. Verdoolaege, S., Juega, J. C., Cohen, A., G´omez, J. I., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Trans. Architec. Code Optim. 9, 4, Article 54 (January 2013).
5. Wolfe, M.: More Iteration Space Tiling. In: Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 655-664. Reno, NV, USA (1989).
6. Bernstein, A. J.: Analysis of Programs for Parallel Processing. IEEE transactions on electronic computers, vol. EC-15, No. 5 (October, 1966).
7. Doroshenko, A., Shevchenko, R.: TermWare: A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae 72(1-3) (2005).
8. GitHub Repository, https://github.com/o-beketov/matmul
9. Aarseth, S. J.: Gravitational N-body simulations. Cambridge University Press (2003).