*Proceedings of the 10th International Conference*
*on Computational and Mathematical Methods*
*in Science and Engineering, CMMSE 2010*
*27–30 June 2010.*

# Automatic Data Layout at Multiple Levels for CUDA

**Yuri Torres**[1], **Arturo González-Escribano**[1] **and Diego R. Llanos**[1]

[1] *Departamento de Informática, University of Valladolid*

emails: `yuri.torres@alumnos.uva.es`, `arturo@infor.uva.es`, `diego@infor.uva.es`

**Abstract**

Trasgo is a source-to-source compiler system that translates simple high-level specifications of parallel algorithms to lower-level native programs, with data partition and communication details generated automatically. Hitmap is the run-time library used by the back-ends of Trasgo for hierarchical tiling and mapping of arrays, currently built on top of the MPI message-passing interface. Hitmap includes a plug-in system for automatic data-layouts. In this paper we extend Hitmap with a new type of data-layout techniques suitable for the CUDA parallel programming model. The combination with the previous type of data-layout techniques allow to generate data distributions, at multiple levels of parallelism, for GPU clusters. The new Hitmap version hides to the programmer the details about the machine structure and thread management, allowing to easily generate programs with multiple levels of parallelism in heterogeneous systems. This work opens the road to develop a new back-end for the Trasgo compiler system to automatically generate CUDA programs.

*Key words: Data layout, CUDA, GPUs, heterogeneous systems*

## 1 Introduction

### 1.1 The Hitmap run-time library

Trasgo [GL09] is a parallel programming system based on high-level and nested-parallel specifications. It provides a C-like front-end language with nested-parallel coordination extensions. The front-end language allows to easily represent abstract specifications of parallel algorithms, with no detail about threads management or inter-process communications. It uses a common scheme to express hierarchical combinations of data- and task-parallelism. The high-level coordination language provided by Trasgo is translated internally to an XML intermediate representation, to allow easier data-flow analysis and code rewriting. Different back-ends may translate the result to native code using different parallel tools or models. Currently, Trasgo have a complete back-end that efficiently exploits the MPI message-passing interface.

The Trasgo back-ends are supported by a runtime library for hierarchical tiling and mapping of arrays, named Hitmap. This library also includes a plug-in system of modules for automatic creation of virtual topologies, and data-partition and layout. The modules are invocated in the code, but applied at run-time with architecture and topology information supplied by the underlying system. Moreover, the resulting layout objects contain all the information needed to map data to the local processor, and to find neighbors on the virtual topology which have data affinities. Thus, the programmer never reasons in terms of system resources, and does not need to know the implementation details of the partition, scheduling, or communication.

Virtual topology functions identify the hosts or cores that are available for the parallel code execution, and use the available topology information to generate a mapping function. Layout functions use the virtual topology information and the index domain of a data structure to generate tiles of the proper grain size for the virtual processors. Hitmap includes several virtual topology functions (such a parallelepiped multidimensional topologies with different restrictions), and several data-layout functions (such as blocks, cyclic, exponential distributions, or dynamic workload balancing of weighted tasks). These techniques work for any special circumstances. For example, they do not need the data elements to be a multiple of the number of virtual processors, and they automatically may assign groups of processors to single data elements if necessary.

## 2 CUDA programming model

CUDA [NBGS08, NVI10] was introduced by NVIDIA to exploit the parallel compute engine in NVIDIA GPUs. Although, CUDA design approach is appropriate for efficient GPU programming, it is conceived as a general purpose parallel computing model. However, there are important differences with other popular parallel computing models, such as message-passing, OpenMP, or PGAS. CUDA works with a shared-memory architecture model. In other shared-memory programming models, such as OpenMP, each task is expected to be launched in an independent CPU core. The memory hierarchy is hidden, and the programmer typically takes into account the number of cores to produce coarse-grain computations to process data in big memory chunks. On the other hand, in CUDA each task will be launched in a SM (streaming multiprocessor) composed by a fixed number (eight) of cores. Inside the SM, the computation model is SIMD (Single Instruction, Multiple Data). Each SM has its own small shared memory, and synchronization system. All SMs in the same device (GPU card) share a bigger global memory. Several devices may work in parallel, receiving, processing and returning data pieces to the main host memory. CUDA places on the programmer the burden of managing the memory hierarchy, and taking decisions about the how to organize the fine grain synchronized tasks which are grouped and pipelined through the streaming multiprocessors. Practical experience shows that this approach is often tedious and error-prone, needing abstractions to hide details and help the programmer [HA09]. Moreover, working with several GPU devices in parallel adds another level of complexity.

## 3 Data-Layout combinations at multiple levels

In previous versions of Hitmap, the run-time system was oriented to create coarse-grain data partitions. It was designed for efficient SPMD implementations in programming models based on interprocess communication, such as message-passing. In this work we extent the functionalities of Hitmap, to support combinations of multiple levels of coarse-grain, and fine-grain layouts.

Consider a system with several GPU cards. Although, the synchronization and communication system across them is limited, we may describe the topology of GPU devices. In CUDA, it is possible to automatically obtain information about the current GPU devices at run-time. Thus, the current Hitmap topology functions and coarse-grain data-partition modules are perfectly suitable to automatically distribute computations, across several GPU devices, with coarse-grain techniques.

However, inside the GPU device we have a different level of parallelism. The computation pieces should be distributed with a different, fine-grain, approach. In CUDA, the number of SPs and SMs in a device is not as important as the number of threads supported by a single SM. Threads are grouped in packs which are executed in the same SM. Groups are pipelined through the processing elements. Thus, threads are grouped and executed in a two-level nested-parallel model. The threads on each group share a local memory, and all groups share the device global memory. Communication or synchronization across groups is possible with atomic operations on the global memory. Thus, the identification of the group and thread are relevant for the computation, not the identification of the processing element.

We define a $\mu$layout (or *ulayout*) as a function to make a domain partition in terms of the number of data elements to be processed together, instead of the number of processing elements (number of SPs and SMs). The output of a $\mu$layout function is a structure of groups of domain elements. Each group will have an appropriate number of domain elements to be processed as a block, and CUDA will be responsible of the assignment of each block to the corresponding SMs. Restrictions to the group size, or shape, may be also imposed by the application, by other $\mu$layout results, by the SM local memory limits, or by the programming model itself (in CUDA the maximum number of threads in a block is limited to 512). We define the first $\mu$layout functions for multidimensional blockings, or cyclic assignment of elements. The Hitmap plug-in system for classical layouts has been replicated for $\mu$layouts. Programmers may add new $\mu$layout functions as modules.

The output of a $\mu$layout may be used by the Hitmap coarse-grain layout functions to distribute the groups across several processing elements or devices. A parallel computation may be deployed on a GPU system using: (1) the new Hitmap $\mu$layout functions to adapt the computation grain and the distribution of the data to the internals of the GPU device, and (2) the topology and layout functions to distribute sets of medium-grain computations across several devices. Thus, a good data-locality may be achieved at the lower level, and a good load balance at the higher level.

## 4 Conclusions

Hitmap is a run-time library for hierarchical tiling, and automatic mapping of tiled arrays. It is designed to support code generation by the back-ends of Trasgo, a source-to-source compiler system.

This work introduces a new type of data-layout techniques into Hitmap. Previous techniques in Hitmap focused on the automatic creation of coarse-grain data distributions in terms of the underlaying machine topology information. The new type focuses on the creation of groups of fine-grain computations to map into a GPU device. The combination of both types allow to develop multiple levels of automatic data-layout techniques for heterogeneous systems in CUDA. This work opens the possibility to develop a Trasgo back-end to generate efficient CUDA programs.

## Acknowledgements

This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302), Ministerio de Industria, Spain (FIT-350101-2007-27, FIT-350101-2006-46, TSI-020302-2008-89, CENIT MARTA, CENIT OASIS), Junta de Castilla y León, Spain (VA094A08), and also by the Dutch government STW/PROGRESS project DES.6397. Part of this work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme. The authors wish to thank the members of the Trasgo Group for their support, and Dr. Valentín Cardeñoso-Payo, and Prof. Arjan van Gemund, for many helpful discussions during the early stages of this research.

## References

[GL09] Arturo González-Escribano and Diego Llanos. Trasgo: a nested-parallel programming system. *The Journal of Supercomputing*, 2009.

[HA09] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[NVI10] NVIDIA. NVIDIA CUDA ProgrammingGuide 3.0. 2010.