Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

SEBASTIAN DRESSLER, THOMAS STEINKE

A Novel Hybrid Approach to Automatically Determine Kernel Interface Data Volumes

This work is funded by the German BMBF project ENHANCE, grant no. 01IH11004A-G.

ZIB-Report 12-23 (July 2012)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

A Novel Hybrid Approach to Automatically Determine Kernel Interface Data Volumes

***Abstract*—Scheduling algorithms for heterogeneous platforms make scheduling decisions based on several metrics. One of these metrics is the amount of data to be transferred from and to the accelerator. However, the automated determination of this metric is not a simple task. A few schedulers and runtime systems solve this problem by using regression models, which are imprecise though. Our novel approach for the determination of data volumes removes this limitation and thus provides a solution to obtain exact information.**

I. Introduction

For heterogeneous systems, schedulers and runtime systems like those from Beisel et al. [1] and Pienaar et al. [2] require several runtime information to select the optimal target accelerator for a kernel. Here we denote the term kernel as one or more functions of an algorithm which are best suited for acceleration. The two most important of these information are the expected wall-clock time of the kernel and the data volumes to be transferred to and from the accelerator device.

The named metrics are useful for determining whether the runtime of the kernel justifies the required data transfer, which consumes additional non-computational time. However, the difficulty is originated by the acquisition of these metrics. Pienaar et al. [2] developed a runtime system for scheduling heterogeneous tasks using parallel-operator directed acyclic graphs (PO-DAGs). This runtime system implements among others a communication model as well as a kernel model. The kernel model estimates the runtime of the kernel with heuristics of previous kernel runs while the communication model estimates by a regression model the amount of data to be transferred.

The above communication model has two limitations:

1) The result is only an estimate and thus too inexact for kernels where the amount of data to be transferred is critical for scheduling.
2) The approach is not well suited for complex data structures like trees.

We present a novel approach for collecting the amount of data to be transferred from the host to the accelerator and vice versa associated with kernel call on accelerator. We refer to this specific data as *kernel data volumes* (KDV). Our approach does not possesses the discussed limitations and requires only a minimal effort of the application developer for implementation.

The paper is organized as follows. Section II describes the proposed method and tool flow. Within Section III we explain how data structures are analyzed. Section IV provides test results for the tool flow. The related work is given in Section V. In Section VI we provide a conclusion as well as an outlook.

II. Method

To overcome the limitations of a priori estimates of KDV, we have to reflect in which state of an application the exact amount of data is known. The sole state where an exact statement of KDV can be made is during the applications runtime prior the call of the kernel function. Therefore, the central question is how one can extract the KDV information of such a pending kernel call.

A simple approach is to make the application developer implement an additional function collecting and returning the current KDV. However, this is not a feasible solution since it tends to be error-prone due to the manual modification of the code by the application developer. For example, the application developer could simply forget components of the desired KDV or perform wrong calculations. Furthermore, when porting kernels to heterogeneous architectures with specialized tool flows (e.g. described in [3]) possibly no knowledge regarding the data structures exists.

A more practicable solution is to automatically determine the KDVs for given kernels with an appropriate tool flow. To implement such a tool flow we use the functionality provided by the Low Level Virtual Machine (LLVM, [4]). The tool flow is described within the following sections. It is a hybrid tool flow, since it uses both static and dynamic analysis.

A. Prerequisites and Term Definitions

First we define terms that are used within this paper. Furthermore, we present necessary prerequisites.

1) LLVM IR: For utilization of the LLVM framework the original source code needs to be transformed into an intermediate language LLVM can work with. This language is called LLVM Intermediate Representation (LLVM IR) and is comparable to a machine independent assembly language. A complete overview of the language elements used by LLVM IR is given in [5].

2) Compiler and linker infrastructure: To generate LLVM IR, an appropriate compiler and linker has to be used. For C and C++ applications we use *clang* [6]. Furthermore, the *GNU binutils* linker *ld* provides an appropriate plugin architecture.

3) Directional size variable: To record the sizes of the subjected KDVs we introduce a variable, referred to as *directional size variable* (DSV), since we have to differentiate between host to device and device to host transfers. Thus, whenever we state that a specific amount of data is added to the DSV, the direction of the data transfer is also taken into account.

B. Proposed Tool Flow

1) Kernel annotation and compilation to LLVM IR: The application developer has to annotate all kernel functions,

Listing 1. Kernel annotation example for C code, showing how to insert a special label "kernel" to mark the kernel function, that is visible within LLVM IR after compilation.

```
1 __attribute__((annotate("kernel")))
2 <retval> <fname>(<params>) {
3     [...]
4 }
```

which is done with so-called attributes. This style of kernel function annotation is preferred since the annotation is also visible within the LLVM IR. This is not the case for `#pragma` statements, being the reason why these are not used, although they may be more familiar.

An example for kernel annotation for C code is given in Listing 1. The same method is usable for C++ code, since this language is also supported by the clang compiler.

2) Code analysis and transformation: This phase of the tool flow builds the analytical basis and includes a mechanism for injecting code evaluated at runtime. The procedure of this LLVM pass for a single kernel function is described next.

First, we do not differentiate between the return value and the function arguments, and thus consider all of them simply as parameters, and are further represented by a DSV.

Each parameter consists of two distinct elements, namely a parameter type and a parameter value. Both elements are necessary for extracting the KDV. Thus, when the type of a parameter is a basic data type it is possible to directly apply the value of the parameter to the DSV. Examples for basic data types are scalars and characters. If the parameter type consists of a complex data type this specific parameter has to be analyzed in depth. Complex data types are all remaining data types like pointers, strings, structs, classes and trees for instance. We provide a detailed description of this analysis process in Section III.

It is important to note that not all information are available after the static analysis. That is, whenever dynamic structures are present – like pointers – the exact evaluation of their size needs to be made at runtime.

After the analysis step is finished the collected KDV information are summarized and injected into to application. This part is known as code transformation. For this purpose we use the capability of LLVM to inject specific instructions like arithmetic operations and function calls. Thus, a call to a function, which processes the collected KDVs at runtime, is injected right before the corresponding kernel call. With this we ensure that the exact KDV information are provided before they are needed.

The described sequence of code analysis and code transformation is repeated for every additional annotated kernel.

3) Re-compilation and runtime evaluation: When the combined analysis and transform pass is finished the LLVM IR needs to be re-compiled, since several additional functions were injected. In the re-compilation step the binary LLVM IR code is first translated into x86 assembler code and then compiled to the target instructions, e.g. x86 machine code.

III. Parameter analysis

This section describes how various data types are processed by the analysis and transformation pass.

A. Scalars

The analysis of scalars is fairly simple since the data type of a scalar is always of fixed size. Thus, only the bit width of its data type is considered. Within the transformation pass this bit width is added to the analogue DSV.

B. Vectors and Matrices

Regarding this data type we make a clear distinction between the used programming language.

For C, we assume that a vector or matrix is represented as pointer with one or more dimensions. Furthermore, it is assumed that the pointer is allocated exactly once before the kernel function is called. Thus, only the functions *malloc* and *calloc* are used for memory allocation. A reallocation with *realloc* is currently not supported, yet.

If C++ is the used language we assume that the vector or matrix is either implemented as C pointer or by as a container of the standard template library (STL). For the implementation as C pointer the same rules defined prior are valid.

We now describe the analysis of the two different implementations in detail.

1) Pointer based implementation: For this case the parameter is back-traced in the IR by using its value which provides a reference to all other uses of the parameter. The back-trace is performed until a call instruction to a *malloc* or *calloc* function is found. This instruction call is decoded and thus it provides the arguments issued to the called function. The further processing depends on the type of the argument of the allocation function:

1) If the argument is a constant, its value is directly used and the DSV is incremented by the value of the variable. Thus, the argument is evaluated statically.
2) If the argument is a variable, the transformation pass injects another call to a size function. This method is used since the content of the variable is not known at compile time. Therefore, the injected size function evaluates the size of the variable at runtime. Here, the argument is evaluated dynamically and thus the DSV is incremented at runtime.

If the pointer points to a multi-dimensional array, multiple calls to the allocation function are discovered. Then, the process of incrementing the DSV or injecting a corresponding function call is repeated.

When the allocation function is called within a loop, the loop is examined to extract the loop counter and end condition. With these additional metrics the increment operation to the DSV is extended accordingly.

2) STL based implementation: Every STL container provides a *size* function. Therefore, we inject a function call that invokes this function of the corresponding object. This call is executed at runtime and its result is used for incrementing the DSV. However, the direct use of the *size* function is only valid

Listing 2. Excerpt of the HD algorithm. Code snippet shows the memory allocation for the matrix with N being the size of the matrix.

```
float **datapoints_g =
    (float**)malloc(N * sizeof(float*));

for (i = 0; i < N; i++) {
    datapoints_g[i] =
        (float*)malloc(N * sizeof(float));
}
```

Listing 3. Annotation of the kernel function of the HD algorithm.

```
__attribute__((annotate("kernel")))
void heat_calc_cpu(
    float **datapoints_g,
    float **datapoints_h,
    int M
);
```

Listing 4. Annotation of the kernel function for the TF algorithm.

```
__attribute__((annotate("kernel")))
void tf(
    int k,
    std::string &genome,
    vector<int> &count,
    vector<string> &kmer
);
```

if the template parameter of the container is a scalar. If this is not the case, a helper function for determining the size is called. For example, with elements of varying sizes, the helper function iterates through all elements of the vector and adds up the size of every element thus returning the size of the whole container.

C. Complex Data Structures

Our definition of complex data structures includes trees and custom classes / structures. These structures are all handled as LLVM structures internally. Thus, all of the named data structures can be analyzed in a similar way.

In the analysis step the data structure is parsed and every element is stored as an internal object. This object contains condensed information regarding the data type of the observed element. These information are, for example, the base type of an element and its bit width as well as an occurrence count.

Additionally, different data types of the structure elements are differently addressed. That is, for scalars the element count can be simply incremented. Pointers and references are traced to its source, thus enabling the analysis of tree structures.

IV. Results

We implemented the proposed tool flow for several basic data structures. Currently it supports scalars, pointers, multi-dimensional arrays and the STL containers `string` and `vector`. For multi-dimensional arrays the loop used for allocation has to be a simple `for` loop. Trees and custom classes are currently not yet supported. We tested the implementation with two different applications. All tests were executed on a 64-bit Linux operating system.

A. Test Case 1: Heat Distribution

The heat distribution algorithm (HD) is implemented in C. It calculates the heat distribution of a single heat source over a fixed period. The algorithm operates on a $N \times N$ matrix, implicating that two-dimensional arrays are used. Therefore, the memory allocation must be done within a loop. We depict the memory allocation for this particular example in Listing 2. In Listing 3 we show the declaration and annotation of the kernel function. To provide a reference measurement of the KDV for the HD algorithm we implemented an appropriate function, which calculates the KDV separated from the code injected by our tool.

Within the analysis pass all of the three parameters were analyzed separately with the method described before. The first two parameters are pointers to two-dimensional arrays, which contain the matrices needed for calculation. Both of these were traced back until calls to *malloc* were found. Within this particular case multiple calls of this type could be detected. Therefore, it was tested whether these calls are enclosed in a loop body. Whenever this was the case, the tool tried to detect the loop counter variable and used it for calculating the resulting memory transfer size. This led to the result that the size of one matrix s_M is given by $s_M = N \cdot 8 \text{ byte} + N \cdot N \cdot 4 \text{ byte}$.

The first part of the allocation shows a different base size since within this case a pointer is allocated. Since the size of N is only known at runtime the tool placed a call to this variable, evaluating its content at runtime.

For the remaining parameter M the tool could identify this variable as scalar integer type with a bit width of 4 byte. The size of this variable simply increments the DSV.

After completion of the tool-flow the application was run with different random sizes of N. The results of both, the KDV calculated with the automated tool-flow as well as the reference measurement, were provided as text output. During all tests the results were always identical, which proves that the tool works with two-dimensional arrays of scalar data types.

B. Test Case 2: Sequence Term Frequency

The sequence term frequency application (TF) is written in C++. The TF algorithm matches several substrings against a single large string and counts the occurrences. We show the interface of this algorithm in Listing 4. For reference measurements a function determining the size of the arguments was implemented.

The analysis pass again parsed all arguments of the function call. For this case, the first parameter was identified as scalar, so its bit width of 4 byte was added to the DSV.

All remaining parameters are considered as containers. Thus, a deep analysis was performed, due to the complexity of this task we further describe every parameter in the following paragraphs.

a) `std::string &genome:` For this parameter the tool detected that it is a string container. Therefore it placed a call to the `size` function of the corresponding variable `genome`. A multiplication operation was placed to ensure that the size of the string is multiplied with 1 byte, being the size of a single character. This computational chain will be evaluated at runtime, providing the current size of the string in bytes.

b) `vector<int> &count:` This parameter belongs to the `vector` class provided by the STL. The in-depth analysis showed that the allocated object of the container is an integer type with a bit width of 4 bytes. Since this type always has a fixed bit width, the tool placed a call to the `size` function of the corresponding variable `count`. Again, a multiplication operator was injected to multiply the result of the `size` function with the bit width of the base type. Similar to the previous parameter this computational chain will also be evaluated at runtime.

c) `vector<string> &kmer:` Regarding this parameter the tool detected that it is a `vector` class, but with a different allocated object. Since the allocated object again is a complex type, it was also analyzed in depth. The analysis showed that the surrounding container could contain strings of arbitrary length. Therefore it is not possible to just call the size function and multiply it with a fixed factor. To solve this conflict the tool placed a call to a helper function. This function iterates over all components of the vector, returning the summarized size of every single string. Since `std::string` uses ASCII characters internally, the size of the string equals the size in bytes.

The results of the analysis and transform pass were evaluated at runtime in addition with the reference size function. We randomly varied the sizes of the parameters. All runs showed equal results for both, automatic estimated KDV as well as reference KDV.

V. Related work

Beisel et al. [1] and Pienaar et al. [2] both reveal the need for automated KDV extraction. However, only in [2] a solution is provided by using a regression model. We exposed that this solution is not feasible in terms of precision, since the regression model only provides estimations for different problem sizes. Furthermore, a history of kernel runs with different KDV must exist, whereas our tool provides direct access to the information.

Another approach is made by Tian et al. [7], presenting a dynamic scheduling mechanism for MapReduce jobs within a heterogeneous environment. Here, the assumption that the scheduled tasks are approximately equally sized. Again this approach is not feasible, since it is limited to MapReduce and furthermore the condition on equally sized tasks is not applicable on generic applications.

Furthermore, Jimenez et al. [8] introduces a predictive scheduling algorithm. A need for KDV could be identified. However, the algorithms implementation makes the developer to explicitly provide the KDV as a distinct variable.

VI. Conclusions and Future Work

Within this paper we presented the idea of automatic determination of the data volume associated with a kernel call. For this, we introduced a hybrid approach consisting of static and dynamic analysis. With this concept study, we proved that our concept is usable and provides correct information.

Our future work concentrates on the support of complex data structures like trees, structs and custom classes. Furthermore, we will focus on the extension of the tool flow being capable of analyzing applications written in FORTRAN.

Acknowledgment

The tested source code of the Sequence Term Frequency and the Heat Distribution application are kindly provided by Daniel Langenkämper (CeBiTec, University Bielefeld) and by Dustin Feld (Fraunhofer, SCAI), respectively.

References

[1] T. Beisel, T. Wiersema, and C. Plessl, "Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux," in *Computer Architecture and Operating System Co-design*, 2012.

[2] J. A. Pienaar, W. Lafayette, and S. Chakradhar, "MDR: Performance Model Driven Runtime for Heterogeneous Parallel Platforms Categories and Subject Descriptors," in *International Conference on Supercomputing*. ACM, 2011, pp. 225–234.

[3] ENHANCE, "BMBF project ENHANCE - enabling heterogeneous hardware acceleration using novel programming and scheduling models," 2011. [Online]. Available: http://www.enhance-project.de/en/project.html 16.06.2012

[4] C. Lattner, "LLVM: A compilation framework for lifelong program analysis & transformation," *Code Generation and Optimization*, no. c, pp. 75–86, 2004.

[5] ——, "LLVM Language Reference." [Online]. Available: http://llvm.org/docs/LangRef.html 16.06.2012

[6] Clang, "clang: a C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org/ 16.06.2012

[7] C. Tian, H. Zhou, Y. He, and L. Zha, "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," in *2009 Eighth International Conference on Grid and Cooperative Computing*. IEEE Computer Society, 2009, pp. 218–224.

[8] V. Jimenez, L. Vilanova, I. Gelado, and M. Gil, "Predictive runtime code scheduling for heterogeneous architectures," *High Performance Embedded Architectures and Compilers*, vol. 5409, 2009.