Beyond Do Loops: Data Transfer Generation with Convex Array Regions

Serge Guelton\textsuperscript{1}, Mehdi Amini\textsuperscript{2,3}, and Béatrice Creusillet\textsuperscript{3}

\textsuperscript{1} Telecom Bretagne, Brest, France
name.surname@telecom-bretagne.fr

\textsuperscript{2} MINES ParisTech/CRI, Fontainebleau, France
name.surname@mines-paristech.fr

\textsuperscript{3} HPC-Project, Meudon, France
name.surname@hpc-project.com

Abstract. Automatic data transfer generation is a critical step for guided or automatic code generation for accelerators using distributed memories. Although good results have been achieved for loop nests, more complex control flows such as switches or while loops are generally not handled. This paper shows how to leverage the convex array regions abstraction to generate data transfers. The scope of this study ranges from inter-procedural analysis in simple loop nests with function calls, to inter-iteration data reuse optimization and arbitrary control flow in loop bodies. Generated transfers are approximated when an exact solution cannot be found. Array regions are also used to extend redundant load store elimination to array variables. The approach has been successfully applied to GPUs and domain-specific hardware accelerators.

Keywords: data transfers, convex array regions, redundant transfer elimination, GPU.

1 Introduction

The last decade has been showcased by the frequency wall limitation and the beginning of a computing era based on parallel computing. One of the solutions that emerges is based on the use of hardware accelerators, for instance Graphical Processing Units (GPUs). These are massively parallel pieces of hardware, usually plugged in a host computer using the PCI-Express bus, that can provide important performance improvements for data-parallel program.

The main drawback of these accelerators lies in their programming model. There are two major points: first the programmer has to exhibit in some way the huge amount of parallelism required to fulfill the accelerator capacity; second, since the accelerator is plugged in the system and embeds its own memory, the programmer has to explicitly manage Direct Memory Access (DMA) transfers between the main host memory and the accelerator memory.

The first point has been addressed in different ways using dedicated languages/libraries like Thrust\textsuperscript{1} with directives over plain C or Fortran [13,26,19].

\textsuperscript{1} http://thrust.github.com/

\textcopyright{} Springer-Verlag Berlin Heidelberg 2013
or through automatic code parallelization \cite{25,6}. The second point has been addressed using simplified input from the programmer \cite{13,27,19}, or automatically \cite{1,26,4} using compilers.

This paper exposes how the array regions abstraction \cite{12} can be used by a compiler to automatically compute memory transfers in presence of complex code patterns. Three examples are used throughout the paper to illustrate the approach: Listing 1.1 requires interprocedural array accesses analysis, and Listing 1.2 contains a while loop, for which the memory access pattern requires an approximated analysis.

This paper is organized as follows: array region analyses are first presented in Section 2; then Section 3 introduces the basis of statement isolation, a compiler pass that transforms a statement into a statement executed in a separate memory space. A redundant transfer elimination algorithm based on array regions is then introduced in Section 4 to optimize the generated data transfers. Finally, some applications are detailed in Section 5.

```
1 // R(src) = \{src[\phi_1] | i \leq \phi_1 \leq i + k - 1\}
2 // W(dst) = \{dst[\phi_1] | \phi_1 = i\}
3 // R(m) = \{m[\phi_1] | 0 \leq \phi_1 \leq k - 1\}
4 int kernel(int i, int n, int k, int src[n], int dst[n-k], int m[k]) {
5     int v=0;
6     for( int j = 0; j < k; ++j )
7         v += src[ i + j ] * m[ j ];
8     dst[i]=v;
9 }
10 void fir( int n, int k, int src[n], int dst[n-k], int m[k]) {
11     for( int i = 0; i < n - k+ 1; ++i )
12         // R(src) = \{src[\phi_1] | i \leq \phi_1 \leq i + k - 1, 0 \leq i \leq n - k\}
13         // R(m) = \{m[\phi_1] | 0 \leq \phi_1 \leq k - 1\}
14         // W(dst) = \{dst[\phi_1] | \phi_1 = i\}
15         kernel(i, n, k, src, dst, m);
16 }
```

Listing 1.1. Array regions on a code with a function call

```
1 // \overline{R}(randv) = \{randv[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}
2 // \overline{W}(a) = \{a[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{5\cdot N+9}{12}\}
3 void foo(int N, int a[N], int randv[N]) {
4     int x=N/4,y=0;
5     while(x<=N/3) {
6         a[x+y] = x+y;
7         if (randv[x-y]) x = x+2; else x++,y++;
8     }
9 }
```

Listing 1.2. Array regions on a code with a while loop