CORE
Metadata, citation and similar papers at core.ac.uk
Provided by INRIA a CCSD electronic archive server

HAL
archives-ouvertes.fr

A unified runtime system for heterogeneous multicore architectures

Cédric Augonnet, Raymond Namyst

To cite this version:

Cédric Augonnet, Raymond Namyst. A unified runtime system for heterogeneous multicore architectures. 2nd Workshop on Highly Parallel Processing on a Chip (HPPC 2008), Aug 2008, Las Palmas de Gran Canaria, Spain. inria-00326917

HAL Id: inria-00326917

https://hal.inria.fr/inria-00326917

Submitted on 6 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A unified runtime system for heterogeneous multicore architectures

Cédric Augonnet and Raymond Namyst

INRIA Bordeaux – LaBRI – University of Bordeaux
cedric.augonnet@inria.fr raymond.namyst@labri.fr

Abstract. Approaching the theoretical performance of heterogeneous multicore architectures, equipped with specialized accelerators, is a challenging issue. Unlike regular CPUs that can transparently access the whole global memory address range, accelerators usually embed local memory on which they perform all their computations using a specific instruction set. While many research efforts have been devoted to offloading parts of a program over such coprocessors, the real challenge is to find a programming model providing a unified view of all available computing units.
In this paper, we present an original runtime system providing a high-level, unified execution model allowing seamless execution of tasks over the underlying heterogeneous hardware. The runtime is based on a hierarchical memory management facility and on a *codelet* scheduler. We demonstrate the efficiency of our solution with a LU decomposition for both homogeneous (3.8 speedup on 4 cores) and heterogeneous machines (95 % efficiency). We also show that a "granularity aware" scheduling can improve execution time by 35 %.

1 Introduction

The last years have witnessed the tremendous invasion of multicore architectures in the field of parallel computing. Although many research efforts have been devoted to designing multicore-aware algorithms and software, application developers are still having a hard time trying to get the most of these hierarchical architectures. Unfortunately, the situation is about to get even worse, with the emergence of a new architecture trend: heterogeneous multicore architectures.

We have known for a long time that making use of specific hardware accelerators can dramatically speed up applications featuring data-parallelism. The novelty is that such hardware is now available off the shelf: clusters featuring GPGPUs, FPGAs or even Cell processors are affordable for most users. It is likely that many supercomputers will be equipped with such heterogeneous hardware in the future. Consequently, programmers now have to deal with architectures composed by a mix of regular CPUs and specific PUs (SPUs, GPGPUs).

Only few applications are currently taking advantage of these hybrid architectures. And even fewer applications are able to use both types of computing units at the same time. Many approaches actually only rely on offloading

parts of the computations over accelerators, since there exists no unified runtime system allowing programmers to seamlessly exploit all available computing units. Thus, the most widely used programming models on such machines are a mix of pthreads and CUDA [3] (for NVIDIA GPGPUs) or LIBSPE (for CELL's SPUs). Of course, higher level approaches exist (RAPIDMIND [14], CELLSS [9], HMPP [8]) that hide the hardware complexity to programmers. However, they currently only support a single type of accelerator at a time. To deal with irregular application, where dynamic load balancing is required, all these systems rely on specific runtime systems that implement memory transfers between main memory and the accelerators.

In this paper, we present the design of a unified runtime system that provides a simple yet powerful interface to exploit multiprocessors equipped with heterogeneous accelerators (e.g. SPU+GPU+CPU). The programming model is based on a high level memory management interface enabling hierarchical description of data domains. Applications tasks are running as "*codelets*" over the underlying hardware computing units: the runtime system automatically takes care of gathering input data before their execution and scattering output results upon completion. Thanks to the concept of divisible tasks, our approach helps to develop programs that dynamically adapt to the available processing units.

2 A uniform approach to exploiting heterogeneous computing units

Designing a runtime system for *heterogeneous* multicore machines, featuring different accelerators and computing units, introduces challenging issues. Traditional shared-memory homogeneous multicore architectures are either programmed using high-level languages (*e.g.* OPENMP [1], CILK [11]) or low-level task management libraries (*e.g.* PTHREAD, LIBSPE). In both approaches, the underlying runtime system is merely devoted to task scheduling. In contrast, heterogeneous machines have much more runtime requirements since they do not provide a coherent (nor even shared) global memory. Unlike regular CPUs that can transparently access the whole global memory address range, accelerators often embed local memory on which they perform all their computations. With no help from the runtime system, programmers would have to explicitly enforce memory consistency between accelerators, which would seriously impact both programmability and portability as we potentially have to deal with various kinds of accelerators (e.g. CELL's SPU with GPGPUs). This definitely emphasizes the need for a runtime featuring high-level memory management interface.

Besides, using the *thread* concept (as provided by modern operating systems) does not allow to efficiently exploit those architectures. Indeed, tasks running over specialized computing units such as GPGPUs can only use a specific instruction set and access a private memory address space. The "*codelet*" concept, which models non-preemptible offloadable tasks, has been recognized to be a much more appropriate execution model [8]. However, provided the variety of accelerator technologies and the gap between their respective performance, stat-

ically determining a relevant granularity is a major issue. Therefore, we believe that the runtime system should offer support for a dynamically adaptive grain size, in collaboration with the programmer who would be responsible for submitting splittable tasks.

2.1 A topology aware hierarchical data management

The major component of our runtime system is a data management facility, which aims at providing a high level API that hides the complexity raised by accelerators' heterogeneity. The idea is to abstract the description of memory regions in such a way that the underlying data transfers and data caching policies can be optimized, depending on the hardware capabilities.

Hardware accelerators typically work only on a subset of data at a time. As it would be unwise to manipulate a huge matrix while only working on a subset of its elements, our library features an interface to manipulate sub-data as well. However, maintaining an MSI protocol on several sub-data is complex as they could overlap. In that case, modifying a piece of data requires to invalidate all copies of the overlapping chunks. This requires to maintain an intersection graph, and makes the coherence protocol NP-complete. Constructing such a graph without an expressive description of data subsets is also costly if not impossible. To address those issues, we introduce the notion of *filter* as a result of two simple observations. First, the programmer is usually well aware of the data layout which the library should therefore not have to infer. Second, we can reasonably make sure all sub-data are disjointed, breaking ties by making new sub-data out of intersections. Contrary to HPF, filters are not restricted to dense matrix structures and can be used with multiple data interfaces (*e.g.* CSR).

```
//use horizontal and vertical n-block built-in filters
filter f1, f2;
f1.func = block; f2.func = vert_block;
f1.arg = n; f2.arg = n;

//declare the data and apply filters on it
data_state *D, *subD;
monitor_blas_data(D, ptr, ld, nx, ny, 4);
map_filters(D, 2, &f1, &f2);

//get a reference to the sub-data
subD = get_sub_data(D, 2, i, j);
// fetch the actual data in local memory
fetch_data(subD, RW);
[...] /* Use subD.ptr for computations */
release_data(subD);
```

Fig. 1. Internal API to manipulate a sub-data

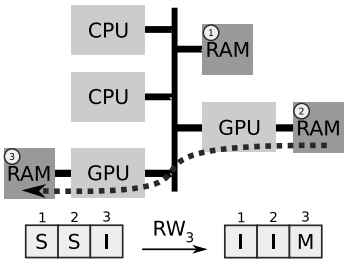

Fig. 2. Coherence protocol

Our runtime features a number of predefined filters (e.g. block-cyclic distribution), and user-provided filters can easily be added if needed. All filters can

be applied recursively, so that data is partitioned following a tree-based hierarchy. Applications must only access the leaves of the resulting tree to maintain consistency of each sub-data independently. This restriction is consistent with a data-parallelism paradigm where all computations are performed locally. When the application needs to access an inner node of the tree, for instance during a *reduction* phase, we temporary unapply some filters. Filters can be efficiently restored later thanks to a lazy design that makes this operation virtually costless.

Once appropriate filters are created and applied by the application, the memory management facility is typically invoked each time a task is scheduled somewhere, to make sure that the required input data is made available on time (see Fig. 1). A straightforward "*write-through*" implementation would simply maintain the up-to-date state of each data in main memory, but this would require numerous unnecessary data transfers between main memory and embedded memory banks. In contrast, the use of a *write-back* model would maintain memory consistency in a lazy fashion (i.e. data transfers are only performed when actually needed) and would thus reduce the stress on the memory buses.

From a technical perspective, implementing such a memory model is challenging, as one cannot assume that it is feasible to directly move data between any combination of accelerators. We thus have extended the pure *write-back* model so that we sometimes allow data modifications to be propagated eagerly to main memory. To enforce data consistency, we maintain a list of the nodes that are holding a valid copy of each data. To this end, we use a similar approach to the one used by *write-back* cache coherency protocols implemented within SMP machines. Since our approach does not rely on any specific hardware support, we use a *directory-based* protocol. As shown on Figure 2, every data is associated to a vector indicating its state on each memory node. There are three states, *invalid* (I) that indicates the node does not hold a valid copy, *modified* (M) for nodes that do have one, and *shared* (S) if there are several nodes with the data. Along with the updating of those states, the MSI automaton also describes which data transfers are needed.

When accessing data for the first time, computing units have to allocate a buffer of sufficient size. The runtime system maintains a reference count of units that actually use the data, so that deallocations are performed in a lazy manner. When a unit accesses a data larger than its remaining memory, some of the unused local data is either discarded or flushed to main memory, depending on the existence of another valid copy of the data. Note that any other unit holding enough free memory is eligible for that purpose. This memory reclaiming mechanism thus allows to transparently handle data sets which size would not entirely fit within accelerators memory. As the size of embedded memory typically vary from orders of magnitude (from 256 KB on a SPU to 1 GB on a GPU), this is crucial when running the same application on various architectures.

2.2 Modeling tasks with codelets

Given the aforementioned restrictions imposed by some hardware accelerators about memory accesses, our unified execution model relies on the use of *codelets*,

which are tasks enriched with a description of their input and output data. This description is performed with the high-level data library we defined in the previous section. While being a constraint for the programmer, the need to precisely specify which data are used opens room for numerous optimizations. It is for instance possible to take data affinity into account while scheduling. Figure 3 illustrates the *codelet* structure used by our runtime. To execute a *codelet*, each eligible accelerator is given a function specific to its architecture. The corresponding "*kernels*" are written by the programmer, but we expect compiling environments to be able to generate them automatically from a generic source code.

The execution of *codelets* is asynchronous and there is no guarantee about their ordering. We therefore added the possibility to perform a *callback* function on the host after the termination of a *codelet*. If synchronization is needed, these callbacks can perform interactions between accelerators and the host. Such interactions can involve costly operations that may decrease the entire system performance, possibly overwhelming the actual *codelet* computation time. The callback mechanism is therefore not sufficient to enforce task dependencies. Hence, our runtime features facilities to express dependencies between tasks within the *codelet* structure itself. This makes programming easier and gives more freedom to the scheduler. It is indeed simpler to extract parallelism when the runtime has a wider view of the task and data dependencies.

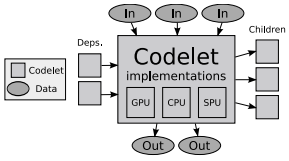

Fig. 3. The codelet structure

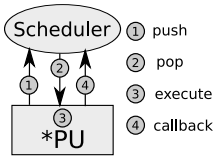

Fig. 4. Execution model

Adding a driver to support a new accelerator architecture only requires to write a limited number of functions as shown on Figure 4. First, the driver needs to supply methods to *fetch* and to *push codelets* from the scheduler which is a mere list of *codelets* from the driver's perspective. This is straightforward on a CPU or a GPU that is controlled directly from the host, but it requires a little more work on a Cell's SPU which has to use DMA mechanisms to manipulate the *codelet* list. The driver must also supply a method that executes the code associated to its architecture in the *codelet* structure. This may involve to actually upload the executable code into the memory of the accelerator, possibly using our data library. Last, the driver needs to offer a mechanism to schedule the execution of a *codelet*'s callback on the host after its termination. On the Cell, we use special hardware interrupts from the SPUs to handle a *codelet* termination and execute its callback.

2.3 Discussion

With respect to our portability requirements, supporting a new architecture is relatively simple. In addition to the limited number of methods needed to execute and manipulate *codelets* on an accelerator, we have to write functions that perform the actual memory transfers with main memory. As memory bandwidth is a scarce resource, we otherwise believe that it is worth writing some optional methods to transfers data directly between some pairs of accelerators, even though there is a slight risk for a combinatorial explosion as the number of supported architectures grows. Provided that our only assumption is that an accelerator needs a mechanism to access main memory, we are confident that our model is well suited for most current and future heterogeneous architectures.

We also believe that our execution model should be helpful for various purposes. On the one hand, the expressiveness of our high-level interface helps programming experts to transmit useful indications to the scheduler while getting some feedback in return. On the other hand, although our solution does not address the code generation problem, compilers can infer most task dependencies, and generate *codelets* that our runtime can schedule efficiently.

3 Experiments

We perform our experiments on a E5410 Xeon quadcore running at 2.33 GHz with 4 GB of memory. This machine also has a nVidia Quadro FX4600 CUDA-enabled graphic card with 768 MB of embedded memory.

3.1 Programmability

Our first contribution is to ease the programming of heterogeneous architectures by the mean of high-level abstractions. This is illustrated on the common matrix multiplication example which follows. After partitioning data into blocks as shown on Figure 1, we actually launch *codelets* as shown below.

```
for (i = 0; i < nslicesx; i++) {
  for (j = 0; j < nslicesy; j++) {
    codelet *cl = malloc(sizeof(codelet));
    cl->where = ANY;   cl->core_func = core_mult;
                       cl->cublas_func = cublas_mult;
    cl->cb = callback_func; cl->argcb = &jobcounter;
    cl->nbuffers = 3; [...]
    cl->buffers[2].state = subdata_ref(&C, 2, i, j);
    cl->buffers[2].mode = W; /* write results into submatrix C_ij */
    push_task(cl); /* schedule the codelet */
  }
}
```

Writing the actual blocked parallel matrix multiplication is then immediate as each *codelet* computes one block of the output matrix. A codelet takes A_i and B_j as inputs and writes its result into $C_{i,j}$, which is described naturally with our high-level library. As there are no task dependencies, the callback only decrements a task counter until there is no more work. Those *codelets* can either be executed on a GPU by calling `cublas_mult` or on a CPU with `core_mult`.

```
void cublas_mult(buffer_descr *descr, void *arg) {
  cublasSgemm(..., descr[2].nx, descr[2].ny ...);
}

void core_mult(buffer_descr *descr, void *arg) {
  cblas_sgemm(..., descr[2].ny, descr[2].nx, ...);
}
```

The programmer must write the kernels that run on the various resources. But the `descr` array is automatically filled with a description of all buffers : the first matrix is for instance located at address `descr[0].ptr`. All the underlying data transfers are transparent, making it much easier to concentrate on the algorithm itself. It is also worth noting that the programmer does not need to take into account the possibility of solving a problem larger than GPU memory.

3.2 Heterogeneous computing

We now validate our core affirmation that it is possible to efficiently use multiple heterogeneous resources by comparing the performance depending on the available computational resources. This experiment consists in multiplying two 16384×16384 single precision matrices. We compute a synthetic performance metric out of the measured execution times.

Table 1 not only shows our system performs well with multiple homogeneous cores as we get a 3.8 speedup on four cores, but we also transparently exploit an hybrid architecture, only supplying the implementation of the kernel on various resources. It appears that we need to devote a core to control the accelerators. On the one hand, the host must remain reactive to external events; on the other hand, such events are especially harmful for cache-sensitive computations such as BLAS. Our runtime system therefore obtains 82.47 GFLOPS with three cores and a GPU, which is 95 % of the added performance of either three cores (25.24 GFLOPS) and a single GPU (62.06 GFLOPS).

Table 1. Combining heterogeneous resources

	1 core	3 cores	4 cores	4 cores / 1 GPU	3 cores / 1 GPU	1 GPU
GFlops	8.70	25.24	32.83	62.34	82.47	62.06

3.3 Extracting enough parallelism

We implemented the LU decomposition presented on the Algorithm 1 twice to underline the importance of a collaboration between the programmer and the runtime system in order to improve scheduling. More precisely, those two versions share the same implementation of the kernels, but the callbacks differ in their handling of the dependencies between *codelets*. In the first version, the

Algorithm 1: Blocked LU decomposition of matrix M

```
for k = 1 to n do
    Decompose M_{k,k} ;                                  /* 1 Codelet A */
    for i = k+1 to n do
        Find M_{i,k} with M_{k,k} M_{i,k} = M_{i,k} ;    /* (n-k-1) Codelets B */
    for j = k+1 to n do
        Find M_{k,j} with M_{k,j} M_{k,k} = M_{k,j};     /* (n-k-1) Codelets C */
    for i = k+1 to n do
        for j = k+1 to n do
            M_{i,j} -= M_{i,k} M_{k,j};                  /* (n-k-1)^2 Codelets D */
```

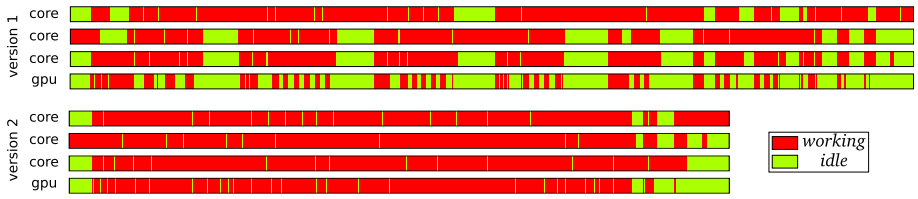

Fig. 5. Supplying enough parallelism helps to reduce load imbalance

codelets of type D are not scheduled until all B and C *codelets* are finished, and the A *codelet* waits for all D ones. This means that the entire algorithm suffers from a sequential section, which is critical for performance. In the second implementation, *codelets* are scheduled as soon as data dependencies are verified. This has a clear effect on the amount of parallelism of the overall algorithm as shown on Figure 5 : while most resources are almost stalled during the execution of the first implementation, they keep running nearly all the time in the second version, substantially reducing the execution time by 15 %.

Table 2. Optimizing the LU decomposition

Optimisation	Reference	Memory pinning	Dependencies	Priorities
GFlops	49.65	53.62	64.94	67.33
Gain (%)	0	8.0	30.9	35.6

There remains load balancing issues at the end of the execution which result from algorithm's inherent lack of parallelism. To avoid that, we added the notion of priority tasks to schedule *codelets* of type A – and all their direct dependencies – as soon as possible. In addition to the use of registered memory that reduces

the need for costly memory copies within the CUDA driver, we see another 5 % improvement using priorities on Table 2.

4 Related work

Programmable GPUs can be controlled using specific languages (*e.g.*, Cg, HLSL or GLSL). Others require less knowledge of graphic APIs using higher-level abstractions like *streams* instead of graphical primitives (*e.g.* Brooks, Scout, Glift). Given that successful evolution toward GPGPUs, well summarized by Ownens *et al.* [17], constructors enriched hardware with generic features and now provide generic programming environments such as CUDA [3], CAL [2]. As summarized by Buttari *et al.* [6], Cell programming usually consists in the use of low-level specific mechanisms provided by the LibSPE, even though there are higher level abstractions either in runtime systems (*e.g.* ALF [7], MCF [4], Charm++ [12], Gordon [15]) or at compile time (*e.g.* Octopiler).

There are also substantial efforts to develop hybrid programming models which differ in the objects they manipulate. On the one hand, there are *data parallel* approaches which map operations on arrays or matrices (*e.g.* RapidMind [14], Brooks [5], PeakStream). On the other hand, some follow a *task parallelism* model, and offer architecture independent abstractions for offloadable functions (*e.g.* Merge [13], Sequoia [10]). A growing number of compiler frameworks are also intended to offer support for heterogeneous architectures (*e.g.* HMPP [8], EXOCHI [13], R-Stream [20]). Likewise, high level asynchronous stream processing systems such as AETHER's S-Net [19] or the Scalp project [15] rely on support at the runtime level. Given the lack of an actual hybrid programming standard, substantial efforts are done to adapt main standards like MPI [18,16], or OpenMP that CellSs extends to express task and data dependencies [9], which is interesting to generates *codelets*.

5 Conclusion and future work

In this paper, we present a runtime system that transparently handles the coherency of hierarchical data structures over heterogeneous multiprocessor machines. With the support of the *codelet* abstraction to model tasks, we sketch the basis of a generic task scheduling platform. We show that our prototype actually obtains very good performance for non-trivial problems, either on accelerators or on hybrid architectures. Those abstractions indeed offer the opportunity to drive accelerator programming beyond the mere solving of technical issues, hence allowing to concentrate on the algorithmic issues. Another conclusion is the importance of a proper task scheduling, and the need for an expressive interface.

In addition to a seamless use of multiple GPU, we plan to make our data library asynchronous which should be profitable in the context of scheduling policies using data prefetching. Besides improving the on-going support of the Cell, we will investigate *lock-free* protocols to prevent scalability concerns. On a longer term, we envision to supply a set of scheduling policies covering a wide

spectrum of problematics, going from a better support of NUMA machines to an inter-node scheduling using MPI. We claim that our runtime could provide the necessary support that compilation environments and specialized libraries lack to harness the growing complexity of heterogeneous machines. With performance portability as a major goal, and given the urgent need for a standardization of all the work around heterogeneous multicore programming, our runtime could contribute to the efforts made around OpenMP and other high level approaches.

References

1. http://www.openmp.org/.
2. AMD FireStream SDK. http://ati.amd.com/technology/streamcomputing/.
3. Cuda zone. http://www.nvidia.com/cuda.
4. B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prelle. Multicore framework: An api for programming heterogeneous multicore processors. In *STMCS*. Mercury Computer Systems, 2006.
5. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH'04*.
6. A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the playstation 3. Technical report, UTK, 2007.
7. C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF '08*, 2008.
8. R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.
9. A. Duran, J. M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependant tasks. In *IWOMP Proceedings*, 2008.
10. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Young Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Supercomputing*, 2006.
11. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98*.
12. D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *PMUP'06*, 2006.
13. M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, 2008.
14. M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. 2006.
15. Maik Nijhuis. *Simple and Efficient Parallel Streaming Applications (working title)*. PhD thesis, Vrije Universiteit Amsterdam, 2008. to appear.
16. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1), 2006.
17. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 03 2007.
18. S. Pakin. Receiver-initiated Message Passing over RDMA Networks. In *IPDPS'08*.
19. Frank Penczek. Design and Implementation of a Multithreaded Runtime System for the Stream Processing Language S-Net. Master's thesis, Institute of Software Technology and Programming Languages, University of Lübeck, Germany, 2007.
20. B. Meister R. Lethin, A. Leung and E. Schweitz. R-stream: A parametric high level compiler. In *HPEC*, 2006.