Resource-agnostic programming for many-core microgrids [1]

T.A.M. Bernard, C. Grelck, M.A. Hicks, C.R. Jesshope, R. Poss

University of Amsterdam, Informatics Institute, Netherlands
{t.bernard,c.grelck,m.a.hicks,c.r.jesshope,r.c.poss}@uva.nl

Abstract. Many-core architectures are a commercial reality, but programming them efficiently is still a challenge, especially if the mix is heterogeneous. Here granularity must be addressed, i.e. when to make use of concurrency resources and when not to. We have designed a data-driven, fine-grained concurrent execution model (SVP) that captures concurrency in a resource-agnostic way. Our approach separates the concern of describing a concurrent computation from its mapping and scheduling. We have implemented this model as a novel many-core architecture programmed with a language called μTC. In this paper we demonstrate how we achieve our goal of resource-agnostic programming on this target, where heterogeneity is exposed as arbitrarily sized clusters of cores.

Keywords: Concurrent execution model, many core architecture, resource-agnostic parallel programming.

1 Introduction

Although nowadays necessary, programming many-core architectures is still difficult [5]. Concurrency must be exposed, and often it is also explicitly managed [8]. For example, low-level constructs must be carefully assembled to map computations to hardware threads and achieve synchronisation without introducing deadlocks, livelocks, race conditions, etc. From a performance perspective, any overhead associated with concurrency creation and synchronisation must be amortised with a computation of a sufficient granularity. The difficulty of the latter is under-estimated and in this paper we argue that this mapping task is too ill-defined statically and too complex to remain the programmer's responsibility. With widely varying resource characteristics, generality is normally discarded in favour of performance on a given target, requiring a full development cycle each time the concurrency granularity evolves.

We have addressed these issues in our work on *SVP* (for Self-adaptive Virtual Processor [2]), which combines fine-grained threads with both barrier and

[1] This work is supported by the European Union through the Apple-CORE project, grant no. FP7-ICT-215216.

dataflow synchronisation. Concurrency is created hierarchically and dependencies are captured explicitly. Hierarchical composition aims to capture concurrency at all granularities, without the need to explicitly *manage* it. Threads are not mapped to processing resources until run-time and the concurrency exploited depends only on the resources made available dynamically. Dependencies are captured using dataflow synchronisers and threads are only scheduled for execution when they have data to proceed. In this way, we automate thread scheduling and support asynchrony in operations.

In the context of this paper, where SVP is implemented in a processor's ISA [4], we have efficient concurrency creation and synchronisation, requiring just a few processor cycles to distribute an arbitrary number of indexed threads to a cluster of cores. Moreover, we can tolerate long-latency operations, such as loads from a distributed shared memory, by supporting asynchrony in individual instructions. The mapping of threads to a cluster of cores in our *Microgrid* hybrid dataflow chip architecture is automatic; compiled code can express more concurrency than is available in a cluster, and mismatches are resolved by cores by automatically switching from space scheduling to time scheduling when hardware thread slots are full. Hence, the minimal requirement for any SVP program is a single thread slot, which implies pure sequential execution, even though the code is expressed concurrently. It is through this technique and the latency tolerance that we achieve resource-agnostic code with predictable performance.

The main contribution of this paper is that we show simply implemented, resource agnostic SVP programs that adapt automatically to the concurrency effectively available in hardware and can achieve extremely high execution efficiency. We also show that we can predict the performance of these programs based on simple throughput calculations even in the presence of non-deterministic instruction execution times. This demonstrates the effectiveness of the self-scheduling supported by SVP. In other words, we promote our research goal:

"Implement once, compile once, run anywhere."

Related work SVP's ability to define concurrency hierarchically and its data-driven scheduling brings it close to Cilk [3] and the DDM architecture [9]. SVP differs from DDM mainly in that synchronisation is implemented in registers instead of cache, and that yet unsatisfied dependencies cause threads to suspend. To our knowledge, no previous work has defined performance envelopes on hybrid dataflow architectures based on architectural constraints.

2 The SVP concurrency model and its implementation

In SVP programs *create* multiple threads at once as statically homogeneous, but dynamically heterogeneous *families*. The parent thread can then perform a barrier wait on termination of a named family using a *sync* action. This fork-join pattern captures concurrency hierarchically, from software component composition down to inner loops. A family is characterised by its index sequence, a thread function and the definition of unidirectional dataflow channels from, to

and within the family. Channels are I-structures [1], *i.e.* blocking reads and single non-blocking writes.

We have built an implementation of SVP into ISA extensions of a novel chip architecture called the *Microgrid*, described in more details in [4]. In this hybrid dataflow architecture, SVP channels are mapped onto the cores' registers. Dependencies between threads mapped to the same core share the same physical registers to allow fast communication and when distributed between cores, communication is induced automatically upon register access. The latter is still a low-latency operation since constraints on dependency patterns ensure that communicating cores are adjacent on chip. Implementing I-structures on the registers also enforces scheduling dependencies between consumers and producers. Hence, long-latency operations may be allowed to complete asynchronously giving out-of-order completion on instructions with non-deterministic delay. Examples include memory operations, floating point operations (with FPU sharing between cores) and barrier synchronisation on termination of a family.

Also, the number of active threads per core is constrained by a block size specified for each family or by exhaustion of thread contexts. Additional expressed concurrency is then scheduled by reusing thread contexts non-preemptively. Deadlock freedom is guaranteed by restricting communication to forward-only dependency chains. The dataflow scheduling, in combination with a large number of hardware threads per core, provide latency tolerance and high utilisation of pipeline cycles.

Another key characteristic of SVP is the separation of concerns between the program and its scheduling onto computing nodes. Space scheduling is achieved by binding a bundle of processors, called a *place*, to a family upon its creation. This can happen at any level in the program, dynamically. On the Microgrid, places are clusters of cores implementing an SVP run-time system in hardware.

The Microgrid is targeted by a system language μTC [7] and a compiler that maps this code to the Microgrid. μTC is not intended as an end-user language; work is ongoing to target μTC from a data-parallel functional language [6] and a parallelising C compiler [10].

3 Performance model and results

Our aim in this paper is to show how we can obtain deterministic performance figures, even though the code is compiled from naive μTC code, with no knowledge of the target. We evaluate results from executing a range of benchmarks across a range of problem sizes on clusters of size 1-64 cores. These include both sequential and parallel algorithms with various data access patterns. The results are presented with performance on cold and warm caches.

In order to analyse the performance, we need to understand the constraints on performance. For this we define two measures of arithmetic intensity (AI). The first AI_1 is the ratio of floating point operations to instructions issued. For a given kernel that is not I/O bound, this limits the FP performance. For P cores at 1 GHz, the peak performance we can expect therefore is $P \times AI_1$, the

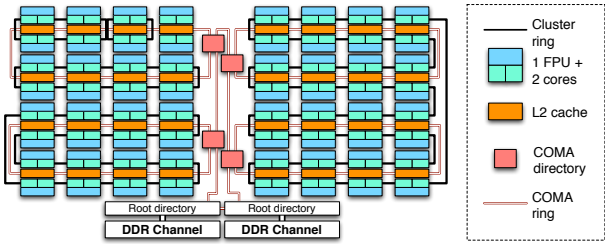


Fig. 1. Functional diagram of our 128 core Microgrid. There are 128 cores sharing 64 FPUs with separate *add*, *mul*, *div* and *sqrt* pipelines. Each core supports up to 256 threads in 16 families using up to 1024 integer and 512 floating-point registers. On-chip memory consists of 32×32KB L2 caches, one per 4 cores. There are 4 rings of 8 L2 caches; the 4 directories are connected in a top-level ring subordinated to a master directory. Two DDR3-1600 channels connect the master directory to external storage. The on-chip memory network implements a Cache-Only Memory Architecture (COMA) protocol [11]: a cache line has no home location and migrates to the point of most recent use. Each DDR channels provide 1600 million 64-bit transfers/s, *i.e.* a peak bandwidth of 25.6GB/s overall; each COMA ring provides a total bandwidth of 64GB/s, shared among its participants; the bus between cores and L2 caches provides 64GB/s of bandwidth; the SVP cores are clocked at 1GHz.

ideal case of full pipeline utilisation (one apparent cycle per operation). In some circumstances, we know that execution is constrained by dependencies between floating point operations, so we modify AI_1 to take this into account giving an effective intensity AI'_1. The second measure of arithmetic intensity is the ratio of FP operations to I/O operations, AI_2 FLOPs/byte. I/O bandwidth IO is usually measured at the chip boundary (25.6GB/s) unless we can identify bottlenecks on the COMA rings (64GB/s). These I/O bandwidths are independent of the number of cores used, so it also provides a hard performance limit.

We can then combine these two intensities to obtain a *maximum performance envelope* for a given code and problem size. A program is either constrained by AI_1 if $P \times AI_1 \leq AI_2 \times IO$ or AI_2 when $P \times AI_1 \geq AI_2 \times IO$.

The results presented in this paper are produced using cycle-accurate emulation of a Microgrid chip (Figure 1) that implements SVP in the ISA. It assumes custom silicon with current technology [4]. It defines all states that would exist in a silicon implementation and captures cycle-by-cycle interactions in all pipeline stages. We have used realistic multi-ported memory structures, with queueing and arbitration where we have more channels than ports. We also simulate the timing of standard DDR3 channels.

3.1 Example: parallel reduction

The first example is the IP kernel from the Livermore suite, which computes the Euclidean norm of a vector. The μTC code is given in Figure 2.

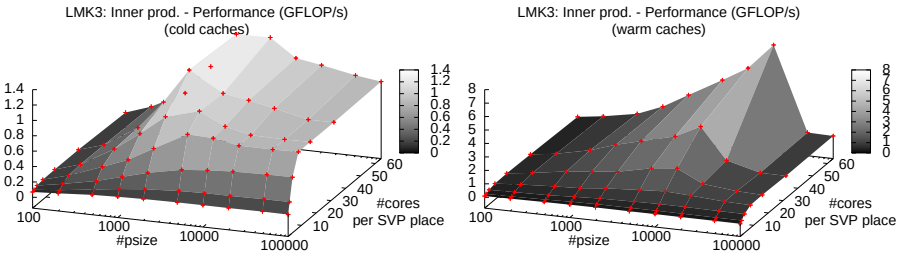
The inner function 'ik3' compiles to 7 instructions including 2 FP operations. So $AI_1 = 2 \div 7 \approx 0.29$. However, every thread must wait for its predecessor to

```
typedef double flt;
/* Livermore loop 3: Inner product
   Q ← ∑_i Z_i × X_i */
thread LMK3_IP(shared flt Q, int N,
               flt Z[N], flt X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}

thread redk3(shared flt Q, flt*Z, flt *X, int sp)
{
   index ri;
   create(LOCAL; ri * span; (ri+1) * sp)
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared flt Q, flt*Z, flt *X) {
   index i;
   Q += Z[i]*X[i];
}
```

Fig. 2. Inner product in μTC using a parallel reduction. Entry point LMK3_IP creates a family at the *default* place, *i.e.* the entire cluster. The family contains P threads where P is the number of cores. Each thread runs 'redk3', identified by 'ri'. Each 'redk3' thread further creates one family of N/P threads running 'ik3'. The keyword 'LOCAL' hints that the concurrency be kept local relative to 'redk3', *i.e.* on the same core if 'redk3' is spread over multiple cores. The reduced sums trickle from the inner family to the entry point through dataflow channel Q.

produce its result before reducing. The cost of communicating the result from thread to thread requires between 6 and 11 cycles per add depending on the scheduling of threads, with the difference representing the cost of waking up a waiting thread and getting it to the read stage of the pipeline, which may be overlapped by other independent instructions in the pipeline. This implies $2 \div (7 + 11) \approx 0.11 \leq AI'_1 \leq 0.16 \approx 2 \div (7 + 6)$, *i.e.* an expected single core performance of 0.11 to 0.16 GFLOP/s. As shown in Figure 3 with P=1 we observe 0.12 to 0.15 GFLOP/s, in accordance with the envelope.



(a) Cold caches (b) Warm caches

Fig. 3. IP performance, using N/P reduction. Working set: 16×#psize bytes.

For multi-core execution we parallelise using commutativity and associativity. The (dynamic) number of cores in the 'current place' is exposed to programs as a language primitive. The reduction is split in two stages: LMK3_IP creates a family of one thread per core, which performs a local reduction and then completes the reduction between cores. When the number of threads per core is

significantly larger than the number of cores, the cost of the final reduction is small and the performance should scale linearly with the number of cores. Given the single core performance of ≈ 0.15 GFLOP/s we would expect a maximum performance of $0.15 \times 64 = 9.6$ GFLOP/s. However, for this code $AI_2 = 0.125$ FLOPs/byte and so performance would be memory limited to 3.2 GFLOP/s.

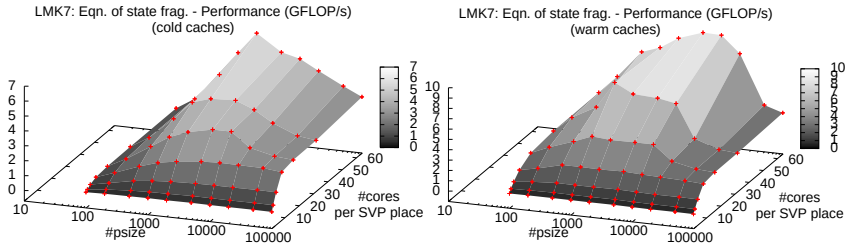
We achieve only 1.4 GFLOP/s, dropping to 0.88 GFLOP/s, for cold caches with the largest problem size. This deviation occurs when the working set does not fit in the L2 caches, because then loads to memory must be interleaved with line evictions. Even though evictions do not require I/O bandwidth, they do consume COMA ring bandwidth. In the worst case a single load may evict a cache line where the loaded line is used only by one thread before being evicted again. A single 8 byte load could require as much as two 64-byte line transfers, *i.e.* a perceived bandwidth for loads of 4 GB/s rising to 32GB/s if all 8 words are used. This translates into a peak performance of between 0.5 and 4 GFLOP/s with $AI_2 = 0.125$ FLOPs/byte, when the caches become full. Note also, at a problem size of 20K on 64 cores, between 17 and 22% of the cycles required are for the sequential reduction, a large overhead and at a problem size of 100K, when this overhead is significantly smaller, only 1/6th of the problem fits in cache for up to 32 cores (1/3 for 64 cores).

With warm caches, this transition to on-chip bandwidth limited performance is delayed and more abrupt. For $P = 32$ the maximum in-cache problem size is N=16K and for $P = 64$, N=32K (ignoring code etc.). As would be expected for ring-limited performance, we see peak performance at N=10K and 20K resp. for these two cases. Any increase in problem size beyond this increases ring bandwidth to the same level as with cold caches.

3.2 Data-parallel code

We show here the behaviour of three data-parallel algorithms which exhibit different, yet typical communication patterns. Again, our μTC code is a straightforward parallelisation of the obvious sequential implementation and does not attempt any explicit mapping to hardware resources. The equation of state fragment (ESF, Livermore kernel 7) is a data parallel kernel with a high arithmetic intensity, $AI_1 = 0.48$. It has 7 local accesses to the same array data by different threads. If this locality can be exploited, then $AI_2 = 0.5$ FLOPs/byte from off-chip memory. Matrix-matrix product (MM, Livermore kernel 21) has significant non-local access to data, in that every result is a combination of all input data. MM is based on multiple inner products and hence $AI_1 = 0.29$. However, for cache bound problems and best case for problems that exceed the cache size, $AI_2 = 3$ FLOPs/byte from off-chip memory. Finally, FFT lies somewhere between these two extremes: it has a logarithmic number of stages that can exploit reuse but has poor locality of access. Here $AI_1 = 0.33$ and for cache-bound problems $1.6 \leq AI_2 \leq 2.9$ (logarithmic growth with problem size if there are no evictions). However, with evictions this is defined per FFT stage and $AI_2 = 0.21$.

For ESF, with sufficient threads, the observed single core performance is 0.43 GFLOP/s, *i.e.* 90% of the expected maximum based on AI_1 for this problem



(a) Cold caches (b) Warm caches

Fig. 4. Performance of the ESF. Working set: 32×#psize bytes.

(Figure 4a). Also, while the problem is cache bound, for cold caches, we see linear speedup on up to 8 cores, 3.8 GFLOP/s. For 8 cores this problem size has 128 threads per core, reducing to 8 at 64 cores. This is an insufficient number of threads to tolerate latency and we obtain 6.6 GFLOP/s for 64 cores, 54% of the maximum limited by AI_2 (12.3 GFLOP/s). As the problem size is increased, cache evictions limit effective I/O bandwidth to 12.3GB/s at the largest problem sizes, *i.e.* an AI_2 constraint of around 6 GFLOP/s. We see saturation at 67% of this limit for both warm and cold caches. With warm caches and smaller problem sizes, greater speedups can be achieved (Figure 4b) and we achieve 9.87 GFLOP/s or 80% of the AI_2 constrained limit for a cache bound problem.

MM naively multiplies 25×25 matrices by 25×N matrices using a local IP algorithm. As $AI_2 = 3.1$ FLOPs/byte, the I/O limit of 75 GFLOP/s exceeds the theoretical peak performance, namely 9.8 GFLOP/s. Our experiments show an actual peak of 8.7 GFLOP/s, or 88% of the maximum.

For FFT, the observed performance on one core is 0.23 GFLOP/s, or 69% of the AI_1 limit. When the number of cores and the problem size increase, the program becomes AI_2 constrained, as now every stage will require loads and evictions, giving an effective bandwidth of 12.3GB/s and as $AI_2 = 0.21$, an I/O constrained limit of 2.6 GFLOP/s. We observe 2.24 GFLOP/s, or 86% of this.

Extra benchmark results are illustrated in Figure 5.

Program	AI_1	AI_2	Bounded by	Max. envelope	Observed	Eff.
DNRM2 (BLAS)	0.14-0.22	0.375	AI_1	0.15-0.22	0.12-0.22	> 80%
MM	0.11-0.16	3.1	AI_1	P×0.16	P×0.13	> 85%
ESF	0.48	0.5	AI_1	P×0.48	P×0.43	> 85%
ESF (cache bound)	0.48	0.5	AI_2	2-6.15 (IO=4-12.3G/s)	2.7	> 40%
FFT1D	0.33	0.21	AI_1	P×0.33	P×0.23	> 65%
FFT1D (cache bound)	0.33	0.21	AI_2	0.84-2.6 (IO=4-12.3G/s)	2.24	> 85%

Fig. 5. Observed performance *vs.* performance envelope for various kernels.

4 Conclusion

The results presented in this paper show high pipeline utilisation of single SVP places by naive implementations of computation kernels. Moreover, we are able to predict performance using a simple performance envelope defined by purely architectural bandwidth constraints. Provided we have sufficient threads we observe performances that are very close (in the region of 80%) to the expected envelope. Even in the worst cases we are within 50% of the envelope.

In other words, the SVP concurrency model facilitates the writing and generation of concurrent programs that need only be written and compiled once but yet can still exploit efficiently the varying parallel resources provided by particular hardware configurations. On our Microgrid architectures programs can be expressed in the μTC language free from the restraints of resource awareness; the program only needs to express the available concurrency in algorithms and the desired synchronisations, and the SVP implementation derives a schedule that achieves high resource utilisation automatically.

References

1. Arvind, Nikhil, S., R., Pingali, K.K.: I-Structures: Data Structures for Parallel Computing. ACM Trans. Program. Lang. Syst. 11(4), 598–632 (1989)
2. Bernard, T., Bousias, K., Guang, L., Jesshope, C.R., Lankamp, M., van Tol, M.W., Zhang, L.: A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors. In: Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008. pp. 1–9 (2008)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., et al.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
4. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. J. Systems Architecture 55(3), 149–161 (2009)
5. Chapman, B.M.: The Multicore Programming Challenge. In: Advanced Parallel Processing Technologies. p. 3 (2007)
6. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.B., Shafarenko, A.: Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computers (CPC'09), Zürich, Switzerland (2009)
7. Jesshope, C.R.: muTC - An Intermediate Language for Programming Chip Multiprocessors. In: Asia-Pacific Computer Systems Architecture Conference. pp. 147–160 (2006)
8. Kasim, H., March, V., Zhang, R., See, S.: Survey on Parallel Programming Model. In: Network and Parallel Computing. LNCS, vol. 5245, pp. 266–275 (2008)
9. Kyriacou, C., Evripidou, P., Trancoso, P.: Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst. 17(10), 1176–1188 (2006)
10. Saougkos, D., Evgenidou, D., Manis, G.: Specifying loop transformations for C2μTC source-to-source compiler. In: 14th Workshop on Compilers for Parallel Computers (Jan 2009)
11. Zhang, L., Jesshope, C.R.: On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In: Bouge, et al. (eds.) Euro-Par Workshops. LNCS, vol. 4854, pp. 38–48. Springer (2007)