Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud Computer

Nicolas Melot, Kenan Avdic and Christoph Kessler
Linköpings Universitet
Dept. of Computer and Inf. Science
58183 Linköping
Sweden

Jörg Keller
FernUniversität in Hagen
Fac. of Math. and Computer Science
58084 Hagen
Germany

***Abstract*—The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. It comprises 48 x86 cores linked by an on-chip high performance network, as well as four DDR3 memory controllers to access an off-chip main memory of up to 64GiB. This work evaluates the performance of the SCC when accessing the off-chip memory. The focus of this study is not on taxing the bare hardware. Instead, we are interested in the performance of applications that run on the Linux operating system and use the SCC as it is provided. We see that the per-core read memory bandwidth is largely independent of the number of cores accessing the memory simultaneously, but that the write memory access performance drops when more cores write simultaneously to the memory. In addition, the global and per-core memory bandwidth, both writing and reading, depends strongly on the memory access pattern.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core "concept-vehicle" created by Intel Labs as a platform for many-core software research. Its 48 cores communicate and access main memory through a 2D mesh on-chip network attached to four memory controllers (see Figure 1).

Algorithm implementations usually make a more or less heavy use of main memory to load data and to store intermediate or final results. Accesses to main memory represent a bottleneck in some algorithms' performance [2], despite the use of caches to reduce the penalty due to limited bandwidth to main memory. Caches are high-speed memories, close to processing units but are rather small and their effect is less visible when a program manipulates a larger amount of data. This leads to the design of other optimizations such as on-chip pipelining for multicore processors [2].

This work investigates the actual memory access bandwidth limits of SCC from the perspective of applications that run on the Linux operating system and use the SCC as it is provided to them. As thus, the focus is not what the bare hardware is capable of, but what the system, i.e. the ensemble of hardware, operating system and programming system (compiler, communication library, etc) achieves. Our approach is to use microbenchmarking to create different sets of patterns to access the memory controllers. Our experience indicates that the memory controllers can support all cores reading data from their private memory, but that the cores experience a significant performance drop when writing to main memory. For both read and write accesses, the available bandwidth is strongly dependent on the memory access pattern.

Section II introduces the SCC, then Section III describes the method used for stressing the main memory interface and discusses the results obtained. Finally Section IV concludes.

II. The Single Chip Cloud computer

The SCC provides 48 independent x86 cores, organized in 24 tiles. Figure 1 provides a global schematic view of the chip. Tiles are linked together through a 6×4 mesh on-chip network. Each tile embeds two cores with their cache and a message passing buffer (MPB) of 16KiB (8KiB for each core); the MPB supports direct core-to-core communication.

The cores are IA-32 x86 (P54C) cores which are provided with individual L1 and L2 caches of size 32KiB and 256KiB, respectively, but no SIMD instructions. Each link of the mesh network is 16 bytes wide and exhibits a 4 cycles crossing latency, including the routing activity.

The overall system admits a maximum of 64GiB of main memory accessible through 4 DDR3 memory controllers evenly distributed around the mesh. Each core is attributed a private domain in this main memory whose size depends on the total memory available (682 MiB in the system used here). Six tiles (12 cores) share one of the four memory controllers to access their private memory. Furthermore, a part of the main memory is shared between all cores; its size can vary up to several hundred megabytes. Note that private memory is cached on cores' L2 cache but caching for shared memory is disabled by default in Intel's framework RCCE. When caching is activated, the SCC offers no coherency among cores' caches to the programmer. This coherency must be implemented through software methods, by flushing caches for instance.

The SCC can be programmed in two ways: a baremetal version for OS development, and using Linux. In the latter setting, the cores run an individual Linux kernel on top of which any Linux program can be loaded. Also, Intel provides the RCCE library which contains MPI-like routines to synchronize cores and allow them to communicate data to each other. RCCE also allows the management of voltage and frequency scaling.

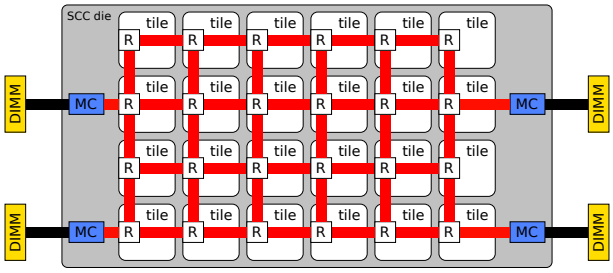


Figure 1. A schematic view of the SCC die. Each box labeled DIMM represents 2 DIMMs.

```
/* SIZE is a power of two
 * strictly bigger than L2 cache
 */
int array[SIZE];

void memaccess ( int stride )
{
  int i, j, tmp;

  for (j = 0; j < SIZE; j += stride)
    tmp = array[j];
}
```

Figure 2. Pseudo-code of the microbenchmark for reading access. For writing, the order of the variables in the assignments is exchanged.

III. Experimental evaluation

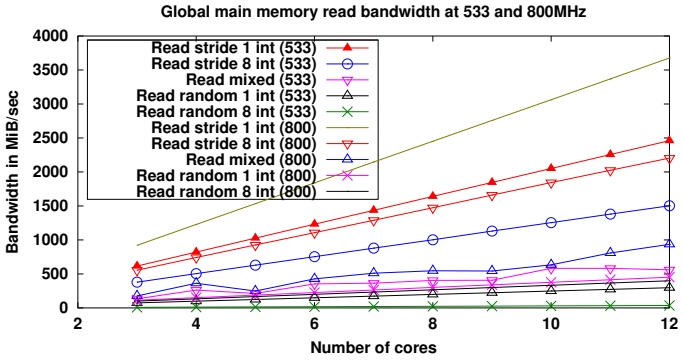
The goal of our experiments consists in the measurement of the bandwidth available to an application that runs on top of the Linux operating system in standard operating conditions (cores at 533 MHz, on-chip network at 800 MHz, memory controllers at 800 MHz). Furthermore, we are interested in how this bandwidth varies with the number of cores performing memory operations and the nature of the operations themselves, read or write. This is achieved by consecutively reading respectively writing the elements of a large array of integers, aligned by 32 bytes which is the size of a cache line. Thus, consecutive access to all integers (1-int-stride, 4-byte-stride) yields perfect spatial locality whereas 8-int-strided access (4 out of 32 bytes) to the data always results in a cache miss. Each participating core runs a process that executes a program as depicted in Fig. 2, where each array is located in the respective core's private memory and through which the cores iterate exactly once.

While the 1-int-strided and 8-int-strided memory accesses stresses the bandwidth difference due to cache hits and cache misses, the *random* access pattern stresses the memory controllers' throughput using a random access, making helpless its hardware optimizations that parallelize or cache read or write accesses, such as using a plurality of open rows in the attached SDRAMs. To simulate random access, the array is accessed through a function $pi(j)$ that is bijective in $\{0,\ldots,SIZE-1\}$, where j is same index (strided 1 int or 8 ints) used to access the array in the strided access described above. In practice, we use $pi(j) = (a \cdot j) \mod SIZE$ for a large, odd constant a where $SIZE$ is a power of two and the size of the array to be read. The random access pattern also applies the 1-int-strided, 8-int-strided and mixed patterns described above to the index j.

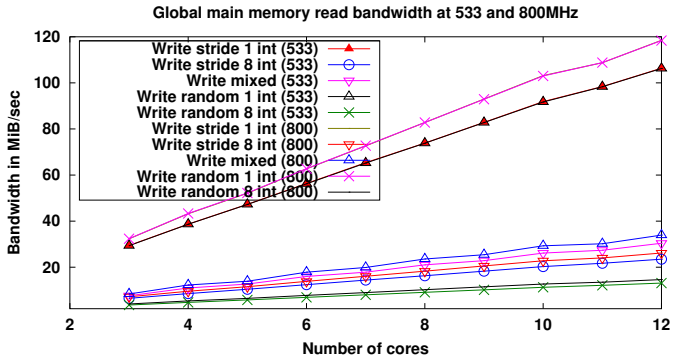
Finally strided, mixed and random access make all the cores read or write at the same time, along the different access patterns they define. All these patterns also combine read and write operations, one half of processors performing reads, and the second half performing writes. This is denoted as the *combined* access pattern.

In this experiment, a varying number of cores synchronize, then iterate through the array to read or write as described above. Since every memory operation leads to a cache miss in the 8-int-strided access and random access reduces the memory controllers' performance, such memory operations generate traffic and the time necessary to read the targeted amount of data allows the calculation of the actual bandwidth that was globally available to all cores. The amount of data to be read or written by each core is fixed to 200MiB. 3 to 12 cores are used, as up to twelve cores share the same memory controller. Cores run at 533 MHz and 800 MHz in two different experiments, while the mesh network and memory controllers remain both at 800MHz. The global bandwidth and the bandwidth per core are measured: the global bandwidth represents the bandwidth a memory controller provides to all the cores. The bandwidth per core is the bandwidth a core gets when it shares the global bandwidth with all other running cores. Figures 3, 4 and 5 show the global and per core bandwidth measured in our experiments.

Figure 3 indicates that both read and write bandwidth are linearly growing with the number of cores. Since the SCC provides no hardware mechanism to manage and share the memory bandwidth served to cores, this shows that all cores together still fail to saturate the read memory bandwidth available. The random access pattern offers a much lower read throughput around 250MiB/sec with 12 cores running at both 533 and 800 MHz. The write throughput for random stride 1 shows the same performance as write stride 1 (up to 105 and 120MHz respectively at 533 and 800MHz) and other write patterns do not exceed 20MiB/sec nor about 7MiB/sec for random stride 8 access pattern. This shows that memory controllers struggle to serve irregular main memory request patterns. The absolute numbers of read bandwidth per core in the 1-int-stride experiment are stable around 205 MiB/s and around 125 MiB/s with the 8-int-stride access pattern with cores running at 533 MHz and respectively 305 and 235 MiB/sec at 800 MHz, as shown in Fig. 4(a). However, the bandwidth per core with the write accesses (Fig. 4(b)) drops with the number of cores from 10 MiB/sec with 3 cores to 9 MiB/sec using 12 cores at 533 MHz and from 11 MiB/sec to 10 MiB/sec at 800MHz. The P54C's L1 cache no-allocate-on-write-miss behavior may explain this performance drop: as write cache misses do not lead to a cache line allocation, every consecutive write results in a write request addressed to the memory controller. In both cases, the low difference

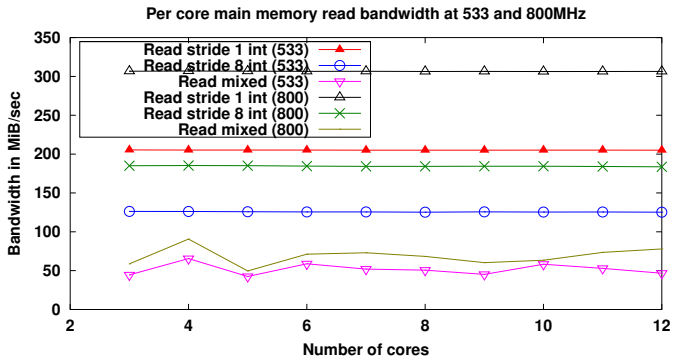


(a) Global main memory read bandwidth at 533 and 800 MHz.

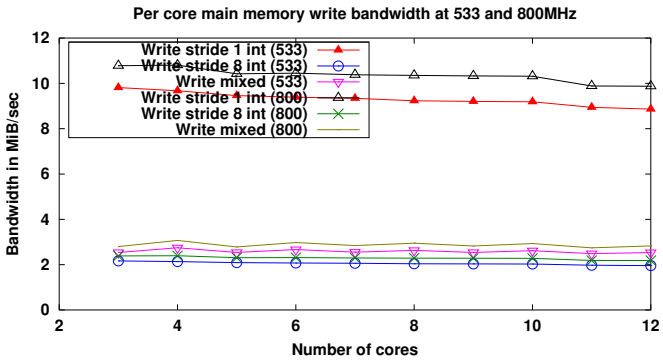


(b) Global main memory write bandwidth at 533 and 800 MHz.

Figure 3. Measured global memory read and write bandwidth as a function of the number of cores involved, at 533 and 800 MHz.



(a) Read memory bandwidth per core at 533 and 800 MHz.



(b) Write memory bandwidth per core at 533 and 800 MHz.

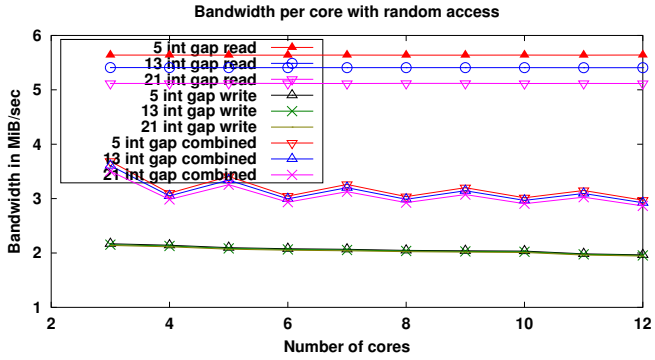
Figure 4. Measured per-core memory bandwidth as a function of the number of cores involved, for strided access patterns, at 533 and 800 MHz.

in performance of 1-int-stride and 8-int-stride access patterns shows that the high performance memory controllers are able to compensate efficiently the performance losses due to cache misses. However the mixed access pattern, with one half of the cores reading memory with a 1-int-stride and the second half with 8-int-stride, exhibits lower performance, which shows again the limited capabilities of memory controllers to serve irregular access patterns.

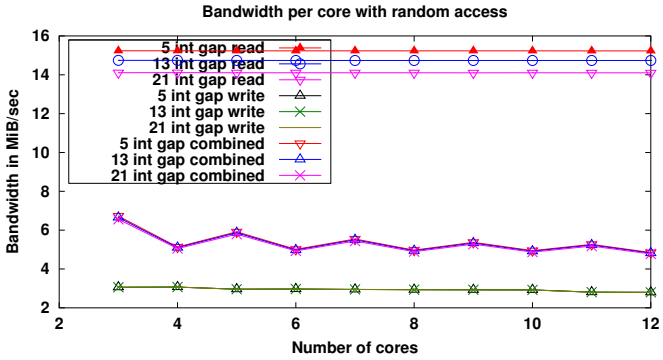
The bandwidth measured per core for the random access pattern reveals better performance with faster cores.

IV. Conclusion

The memory wall represents an important performance limiting issue still present in multicore processors, and implementations of parallel algorithms are still heavily penalized when accessing main memory frequently [2]. This work enlightens the available memory bandwidth on Intel's Single Chip Cloud Computer when several processors perform concurrent read and write operations. The measurements obtained here and the difficulty we experience to actually saturate the read memory bandwidth show that the cores embedded in the SCC cannot saturate all together the read memory bandwidth available: for read access patterns behave regularly, the cores cannot saturate. However, the measurements obtained from the write access patterns demonstrate a much smaller write bandwidth available. Also, we can note that the available bandwidth for both read and write strongly depends on the memory access pattern, as the low bandwidth on random



(a) Memory bandwidth per core with random access pattern at 533 MHz.



(b) Memory bandwidth per core with random pattern at 800 MHz.

Figure 5. Measured per-core memory access bandwidth as a function of the number of cores, for random access patterns, at 533 and 800 MHz.

access patterns indicates. Thus, there is no point in reducing the degree of parallelism in order to increase the available bandwidth for tasks requiring a high main memory bandwidth. The measurements shown in the paper show a behavior possibly adapted to program restructuring techniques such as on-chip pipelining and our previous implementation of on-chip pipelined mergesort [2]. In this implementation, many tasks mapped to several cores fetch input data in parallel from main memory, and a unique task running on a unique core writes the final result back to main memory, therefore limiting expensive main memory accesses. However, the gap between the memory bandwidth available and the limited capabilities of cores to saturate it shows that there is room to add more cores, run them at higher frequency or add SIMD ISA extensions. Without such improvements in the cores' processing speed and accordingly higher demands on memory bandwidth, our ongoing research on program restructuring techniques such as on-chip pipelining is, for SCC, limited to implementation studies leading to predictions of their theoretical speed-up potential, rather than demonstrating concrete speed-up on the current SCC platform. Such techniques could speed up memory-access intensive computations such as sorting [2], [3] on SCC-like future many-core architectures that are more memory bandwidth constrained.

ACKNOWLEDGMENTS

The authors are thankful to Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer". We also thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

This research is partly funded by the Swedish Research Council (Vetenskapsrådet), project *Integrated Software Pipelining*.

REFERENCES

[1] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling," *IEEE J. of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[2] R. Hultén, J. Keller, and C. Kessler, "Optimized on-chip-pipelined mergesort on the Cell/B.E." in *Proceedings of Euro-Par 2010*, vol. 6272, 2010, pp. 187–198.

[3] K. Avdic, N. Melot, J. Keller, and C. Kessler, "Parallel sorting on Intel Single-Chip Cloud Computer," in *Proc. A4MMC workshop on applications for multi- and many-core processors at ISCA-2011*, 2011.