— Author's copy —

Parallel Macro Pipelining on the Intel SCC Many-Core Computer

Tim Süß
Johannes Gutenberg-University Mainz
Zentrum für Datenverarbeitung
Mainz, Germany
t.suess@uni-mainz.de

Andrew Schoenrock
Carleton University
School of Computer Science
Ottawa, Canada
aschoenr@scs.carleton.ca

Sebastian Meisner, Christian Plessl
University of Paderborn
Paderborn Center for Parallel Computing
Paderborn, Germany
{*sebastian.meisner,christian.plessl*}*@uni-paderborn.de*

***Abstract*—In this paper we present how Intel's Single-Chip-Cloud processor behaves for parallel macro pipeline applications. Subsets of the SCC's available cores can be arranged as a pipeline where each core processes one stage of the overall workload. Each of the independent cores processes a small part of a larger task and feeds the following core with new data after it finishes its work.**

Our case-study is a parallel rendering system which renders successive images and applies different filters on them. On normal graphics adapters this is usually done in multiple cycles, we do this in a single pipeline pass. We show that we can achieve a significant speedup by using multiple parallel pipelines on the SCC. We show that we can further improve performance by using SCC's controlling PC in conjunction with the SCC. We also identify aspects of the SCC that hinder the overall performance, mainly the lack of local memory banks for each core on the SCC. The results presented in this paper are not limited to only image processing, but users could expect similar experiences where macro pipelining is used in other applications on the SCC.

***Keywords*-Heterogeneous; Single-Chip-Cloud; parallel; macro pipelining;**

I. Introduction

Since the introduction of multi-core processors to the broad scientific and public computing community, the structure of many programs have changed significantly. Due to the physical limits during the production of a processor and the ever rising temperatures with faster processors, manufacturers now increase the performance and decrease the temperature of their CPUs by increasing the number cores instead of the speed of the individual core. Currently, manufacturers are looking for alternative processor layouts and programming models to increase the overall performance of their chips. Intel's *Single-Chip-Cloud* (SCC) processor is one of these alternatives [1]. This chip is an experimental prototype and consists of 48 independent *Pentium I* cores (P54C). These cores are pairwise arranged on a network grid that enables them to communicate with each other (see Figure 2 or section II for further details). The SCC itself is integrated into a developer kit which also contains an independent *Management Control PC (MCPC)*. This machine controls the SCC like a normal PCIe device and is able to communicate with the different processors over a PCIe bus. Thus, the complete kit forms a heterogeneous computing system. In comparison to GPUs, each processor on the SCC can run its own OS. A benefit of this design is that the SCC cores can be programmed like regular computers. There is no need to learn device specific languages or libraries (like GLSL or CUDA) and there are less limitations in the programmability (e.g. there is no restriction on recursive functions).

Here we aim to use the SCC and the MCPC for parallel macro pipelines. Our goal is to investigate if this is possible and to present the issues involved with doing so, our overall experience using this chip and how our results could potentially be applied to other applications. As a case study we use a rendering application which simulates an old silent film. To this end we used a regular 3D renderer to render a walk through of a 3D scene. Each frame is subjected to a sequence of post processing effects which convert the frame to a sepia-toned monochrome version, blur the resulting image and finally add random amounts of scratches and flickering to create a vintage movie effect. Different configurations are evaluated to analyze the acceleration by this heterogeneous system. In some cases the MCPC is used as a part of the overall workflow, compared to other cases where only the SCC is used. In all configurations we analyze the influence of different alignments of the pipelines within the SCC.

Our contribution: Through our image processing case study, we show that we can use the SCC for macro pipelining which scales if the number of pipelines is increased. Furthermore, we will show that using the SCC in a heterogeneous fashion (in conjunction with the MCPC) we can achieve significant performance improvements. We also identify some challenges in using this system for parallel macro pipelining. In particular, we found that although the SCC's network seems to be designed well to avoid bottlenecks or hotspots, the lack of local memory banks hinder the overall performance. The ideas presented in our work should easily translate to other problem domains where parallel macro pipelines are used. Therefore it is expected that the performance results in this case study of computer graphics on the SCC should be reproducible in other pipelining applications.

Published in: *Proc. Int. Symp. on Parallel and Distributed Processing Workshops (IPDPSW)*
© IEEE 2013, DOI: 10.1109/IPDPSW.2013.136

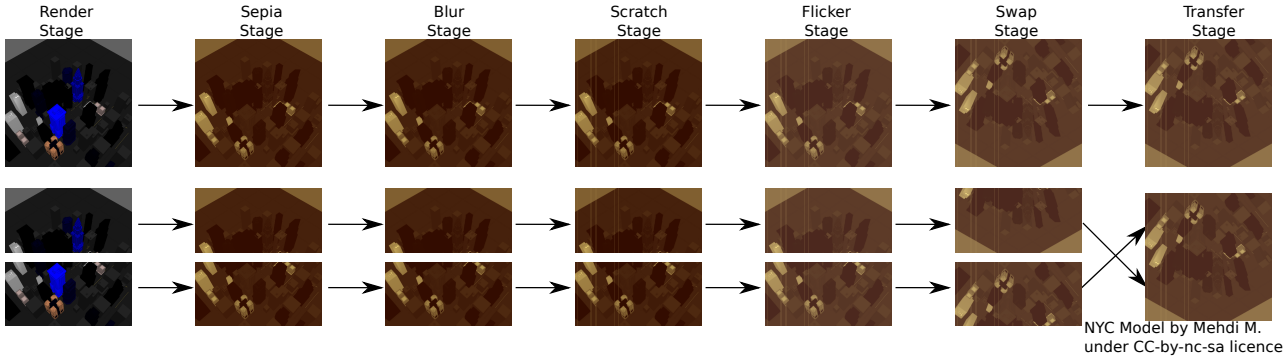

Figure 1. The top row shows the five stages of the image processing pipeline. Each stage is independent from the others and processes its specific task. The bottom row shows the stages of two parallel pipelines processing a single image. Each's node task has become smaller by distributing the complete computation among two pipelines.

II. Overview

Intel's SCC is a multi-processor which consists of 48 independent Pentium I cores (P54C) [2]. The processors' speed can be changed at runtime from 400 MHz (0.7 volts) up to 1198 MHz (1.3 volts) via the RCCE-library [3]. These processors are paired and the pairs are then connected by a 2D grid via routers. The mesh is clocked at a frequency of 800 MHz or 1600 MHz. The routers provide 16 kib memory and each processor is equipped with 16 KiB L1-cache and 256 KiB L2-cache. These caches are 4-way set associative. The individual processors have no local memory, thus all memory accesses must be done through the four memory controllers via the interconnecting grid. The four memory controllers have a maximum memory capacity of 64 GiB DDR3 RAM clocked at 800 MHz or 1066 MHz. Within this global memory each processor has its own private partition, giving no globally shared memory. When the CPUs' clock speed is set to 533 MHz and the RAM and mesh is set to 800 MHz, the SCC consumes approximately 22 watts while idling. The MCPC can communicate with the SCC and vice versa via PCIe or network. The MCPC is equipped with an Intel Xeon X3440 (2.53 GHz) and 4 GiB RAM. While it idles it consumes approximately 52 watts (see Figure 2).

Our parallel macro pipeline consists of a simple 3D renderer, different image manipulating filters, and a transfer stage, which sends the data to a visualization client (see Figure 1). The rendering stage produces an image which is sent through the pipeline stages. At the end of the pipeline, the final images are assembled and sent to the visualization client where it is displayed.

While the visualization client is always placed on the MCPC, in some tests the rendering stage is performed on the SCC and in others on the MCPC. The parallelization of the image generation is done in a sort-first manner [4]. Here the image is split into horizontal strips which are processed by different pipelines autonomously.

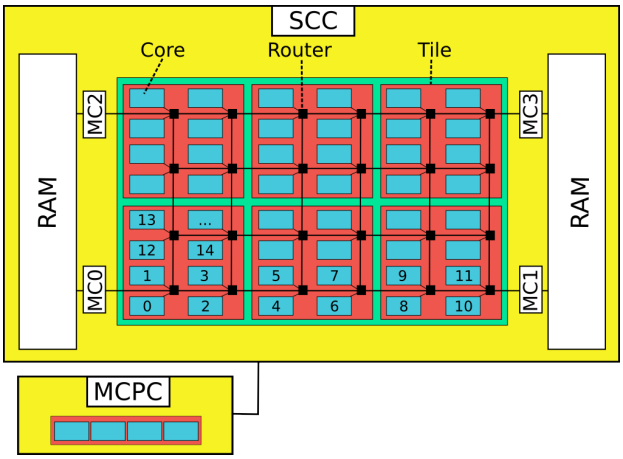

Figure 2. Overview of Intel's *Single-Chip-Cloud* (SCC) processor and *Management Control PC* (MCPC) system.

III. Related work

Modern computers contain pipelines which perform very small, fixed operations. Using processors that provide many cores for macro pipelining is common, even if the cores are heterogeneous. One of the most famous processors of this kind is IBM's Cell processor [5]. It consists of one 64-bit *PowerPC* core and up to eight *synergistic processor elements* (SPEs). It is common to use the SPEs as a ring of macro pipeline stages where each SPE processes a small part of a larger task. Each of these SPEs provides 256 KiB local storage. Dart [6] and MaRC [7] are processors that provide a small local memory with both processors allowing point-to-point connections. Additionally the Dart processor allows communication via a shared memory and the MaRC processor allow multicasts. The PMMLA [8] is another processor which allows macro pipelining.

In comparison to these processors, the SCC's cores do not provide any local memory, only a small cache. Therefore all data must be transferred over the underlying, routed network

grid. This hinders the SCC's usage as macro pipeline. However, due to the large number of regular cores, it allows us to create programs in a relatively conventional manner. Like the Cell processor, the combination of the MCPC and SCC builds a heterogeneous computing system where the two different processor types do not share any memory. The SCC allows the user to write code for the different cores in common C/C++ and not in a domain specific programming language. The shading languages are similar to C but they do not offer all features that are common for most programmers (e.g. pointers).

The classical rendering pipeline on graphic adapters is a macro pipeline similar to ours [9]. In different stages the geometry can be generated and transformed, and the pixel colors are computed. These stages are programmable by different shading languages [10], [11].

Molnar et al. [4] classified the three different classes of parallel rendering techniques: sort-first, sort-middle, and sort-last. The different classes define where in the rendering pipeline the workload is distributed among multiple units. Our system follows the sort-first approach; we split the image into multiple tiles, generate and process them independently. Hardware vendors like ATI or Nvidia provide Crossfire or SLI to distribute the rendering load directly on multiple graphics adapters [12], [13]. Ludic Logix, with their hydra system, allows the combination of ATI and NVidia graphics adapters within one computer [14].

Using heterogeneous systems for parallel rendering can be done in different ways. For example, a large amount of weak nodes can be used to perform image simplifications[15] or visibility tests [16]. In our system, the weak nodes manipulate the images to be displayed.

Schumacher et al. uses FPGAs to compose different sub-images [17]. In contrast to this, the SCC is much easier to program than a FPGA, which simplifies the development.

Many studies on the SCC are done to evaluate the communication capacities [18], [19] or how cache coherence can be achieved [20]. In contrast to our work, these studies use only the SCC for their computations, while we also utilize the MCPC and build a heterogeneous system.

IV. Parallel Macro pipelines

A macro pipeline consists of many independent stages where each stage receives incoming data, processes this data in some way and passes it along to the next stage. In contrast, stages within a traditional pipeline (e.g. in the CPU) where each stage consists of a single operation, these stages perform significant, complex operations. Due to the large number of independent cores provided by the SCC, we parallelized the pipeline to make full use of all available cores on the SCC which, in turn, leads to a significant speedup.

The different stages within the complete image generation and filtering process are relatively easy to pipeline and parallelize. One pipeline consists of different stages: one stage to generate an image from the given CAD data, five stages to manipulate the image, and one to send the final image to a client to display. For the parallelization of this pipeline, we divided the image into several horizontal strips. Thus, the different stages have only to process a smaller part of the overall image. The different stages have different memory access patterns that influence the time needed to apply their operations. The various stages in our pipeline are described below.

Render stage (RS): The renderer generates an image from the input data, which consists of a large amount of colored triangles. When the render stage starts, it loads the scene and organizes the different objects in a hierarchical data structure known as an octree [9]. To determinate the objects placed within the horizontal strip, it performs a frustum culling [21]. By doing this the octree is traversed, causing significant memory accesses. Once the objects within the tile are determined, they must be rendered. Each renderer has its own frame buffer (four bytes per pixel) which stores the pixels of the transformed triangles. After all needed objects have been processed, the resulting image is sent to the next stage and the next image to be processed is rendered. The running time of this stage depends on two parameters: the resolution of the tile to be processed and the complexity of the scene. Overall, this stage could be the most expensive depending on the overall size and complexity of the image. We therefore assume that the rendering task is, in general, the most expensive one.

Sepia stage (SeS): The sepia stage gives the image an older look by changing the overall color. For each pixel, a formula is used which transforms the original color into a shade of brown. This formula is given below where the final color of the pixel is given in rgb_{new}.

$$\begin{aligned} S1 &= (0.2,\, 0.05,\, 0.0) \\ S2 &= (1.0,\, 0.9,\, 0.5) \\ mix &= clamp((0.3 \cdot r) + (0.59 \cdot g) + (0.11 \cdot b)) \\ rgb_{new} &= clamp(S1 \cdot (1.0 - mix) + S2 \cdot mix) \end{aligned}$$

The colors $S1$ and $S2$ are standard, constant colors needed for the sepia tone. First we calculate mix, a number that combines the different color components of a pixel. The different tones are weighted to achieve a brown shift related to pixel's original color. If the mix value is greater than 1.0 (due to floating-point inaccuracies), we set it to 1.0 (what is usually referred as clamping in computer graphics). Lastly, we scale the vectors $S1$ and $S2$ in with respect to the mix value and store this as the new color for the pixel.

Blur Stage (BS): To increase the impression of an old image, we reduce the contrast by applying a blur filter. In this stage the pixels are transformed with respect to the neighboring pixels by calculating the average color of these pixels. To work from the original data, a second buffer is

required to store the results of this transformation. This stage was the most time consuming stage in our test cases. The time spent on this routine depends only on images' resolution, just as the other image filtering stages and could, in some cases, actually take more time than the rendering stage.

Scratch Stage (ScS): To generate scratches in the image, not all pixels must be processed. When this filter begins, two random numbers are chosen: one for the number of scratches and another one for scratch color. Next, for each scratch, an x-coordinate is randomly chosen. On each of these positions the vertical pixels are replaced by the previously chosen color. Our filter provides only vertical scratches but the system can be easily to extended to allow scratches of arbitrary orientation and length.

Flicker Stage (FS): To vary the brightness of individual images within a series of images, we choose a random number in the interval $[-\frac{1}{10}, \frac{1}{10}]$. This value is added to all pixels RGB values and clamped to the $[0, 1]$ interval. Here, each pixel is accessed in sequential order but with a minor operation. When the series of images are viewed sequentially, this operation results in a flickering effect.

Swap Stage (SwS): Our visualization client requires images that have been produced by the pipeline to be vertically mirrored to display them correctly. During the last step, the image is flipped upside-down by swapping appropriate rows of pixels. First line i is copied into an intermediate buffer. Then, the corresponding $j = \sharp lines\, in\, stripe - i$ is copied into line i. Afterwards, the line in the intermediate buffer is copied to the line j. These steps are performed until the image has been flipped. This stage, however, is not necessary. Without it the images will simply be rendered upside down. This stage was included in order to introduce different memory access patterns.

Transfer Stage (TrS): The transfer stage collects the results from the different pipelines, combines them into a complete image and then sends it to the visualization client where it is displayed until a new image arrives. There is always only one transfer stage regardless of the number of pipelines.

Connect Stage (CS): In the case that we use the MCPC during render stage, we need a channel to transfer the generated frames to the parallel macro pipeline. In that case we employ a new stage on one node of the SCC. This stage does nothing besides receiving the frames from the MCPC and distributes them among the pipelines.

A. Pipeline Arrangements on the SCC

In our application, the different cores of the SCC have to access memory simultaneously and not necessarily following a regular pattern. To try to balance the load over the different memory controllers that are provided by the SCC, we tested different arrangements of the pipeline stages. We tested three different alignments: *unordered*, *ordered*, and *flipped*.

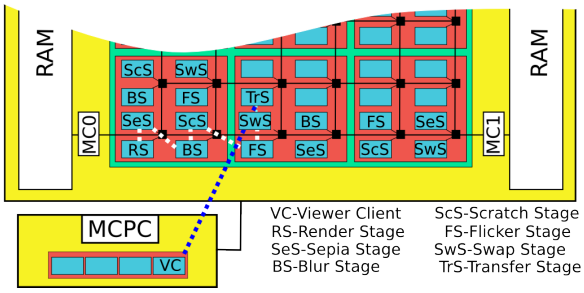

Figure 3. Unordered Arrangement. Pipelines are arranged in a sequential manner. The order is given by the core IDs. Here there is only one render stage that supplies the three different pipelines. Additionally, there is only a single transfer stage which collects the sub images and send them to the visualization client.

The Figures 3, 4, and 5 display three parallel pipelines in different arrangements. Here, only one core is used to produce the images. The dashed, white lines within the images show the data flow of a pipeline.

Unordered Arrangement: In the unordered arrangement we used the processor order given by the SCC. The different, parallel pipelines follow one another in this arrangement (see Figure 3). In this arrangement, the processors of one pipeline are close to each other. However, due to the varying length of pipelines, it is possible that a pipeline will not end before the row of processors on the chip does. In this case a pipeline could start in the middle of a row of processors and end in another row. This would mean that the first stage in the pipeline would not be directly linked with a memory bank and that the pipeline stage to pipeline stage communication may not be simply neighboring processors communicating with one another on the SCC's grid network. It would be expected for this to negatively affect overall performance.

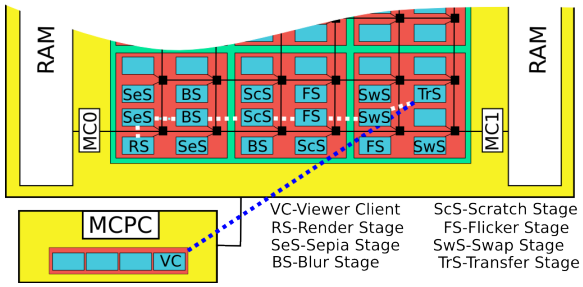

Figure 4. Ordered Arrangement. The (three) pipelines are arranged in a parallel manner along the grid network of the chip with one render stage and one transfer stage.

Ordered Arrangement: In the ordered arrangement we placed the different pipelines in parallel along the network grid on the SCC. In this arrangement we can achieve a one-way communication flow (see Figure 4). Because of this we expect a better utilization of the router buffers for our purposes (due to direct communication pathways), resulting in increased performance.

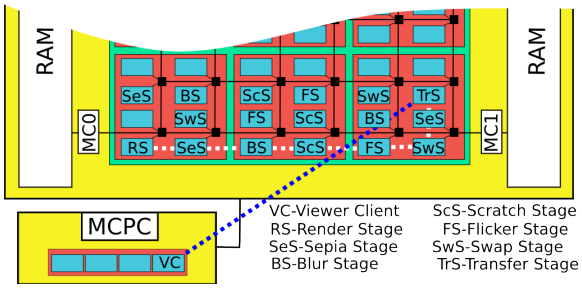

Figure 5. Flipped-order Arrangement. Pipelines are arranged in a parallel manner. The direction of every 2nd pipeline is flipped.

Flipped-order Arrangement: In the flipped-order arrangement, we arranged the pipelines parallel to the grid as well. However, in contrast to the ordered arrangement, we flipped each second pipeline on the SCC (see Figure 5). We expected an additional performance improvement because of better memory controller usage, caused by the flipping the direction of every other pipeline. In this configuration, the memory accesses of the two most costly stages (rendering and blurring) which are placed at pipelines front should be better distributed among the memory controllers. Thus the data accesses should be accelerated.

V. Renderer Configurations

To analyze the performance of the overall rendering and filtering system and to show the benefit of introducing heterogeneity into the system, we implemented three different scenarios. These scenarios are described below. One should note that in all of the scenarios the images are rendered off-screen and the generated and filtered images are sent to the visualization client (running on the MCPC) via UDP. In our tests, we perform a virtual walkthrough through a 3D model. The complete walkthrough consists of 400 individual frames.

One Renderer with Multiple Macro Pipelines: In the first scenario we use only one core of the SCC for the rendering task. By doing this we decrease the massive data accesses caused by multiple renderers and therefore increase rendering load of that single core. Figure 3 shows the ordered pipeline alignment for this configuration. The renderer determines the objects placed in the field of view by performing a frustum culling for the complete screen. Afterwards, these determined objects are rendered and the resulting image is divided into as many strips as pipelines available. These tiles then are sent to the first filtering stage.

As Many Renderers as Pipelines: In our second scenario, we used as many rendering stages as macro pipelines used (see Figure 6). Due to the increase in the number of rendering stages, we cannot create as many pipelines as in the other scenarios. Here each renderer only has to process a smaller portion of the overall image. Distributing the rendering load onto many renderers is known as sort-first rendering. By increasing the number of renderers we increase the total number of memory accesses, however we also decrease the load on each of the rendering processors.

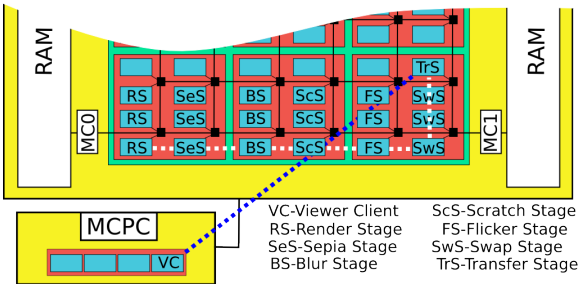

Figure 6. Multiple pipelines are arranged in a parallel manner, each with their own renderer.

MCPC for rendering and many pipelines on the SCC: In our third scenario, we use a heterogeneous rendering system. Since the MCPC mostly idles while the SCC is working, we decided to use its powerful Xeon processor for the most expensive macro pipeline stage: the rendering. While the inner pipeline stages have only to access the contiguous block of memory which corresponds to their tile, the renderer has to access different objects from which the image is made. These objects are organized in a hierarchical data structure. Traversing this data structure is done recursively, which often prevents pre-fetching of the data needed next. On top of this, generating the images is the most time consuming task due to its high computational costs. Since the rendering will be performed off of the SCC, the cores on the SCC only perform the image filtering operations. In this scenario we replaced the render stage on the SCC by a so-called connector stage (see Figure 7). After the renderer has been started it establishes a UDP connection to the connector stage on the SCC. The different rendered images are sent over this channel into the pipelines. As a result of this setup we reduce the necessary memory accesses on the SCC and we make better use of the available resources.

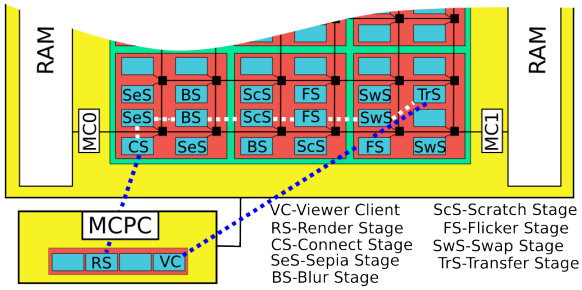

Figure 7. The rendering process is performed on the MCPC while the SCC just filters the frames.

VI. Results

For our implementation we used gcc-4.4.3 on the 64 bit MCPC and gcc-3.4.5 for the SCC cores as our compiler, RCCE-2.0 for our MPI implementation and os-mesa for rendering. The RCCE library is provided by Intel to program the SCC. It is similar to the familiar MPI libraries.

A. Walkthrough time

If we use only a single core to perform the complete pipeline it takes roughly 382 sec for one test run (see Figure 8 for details). This will represent the baseline run time which we will reference to determine the speed-up achieved by the parallel macro pipelines. We will also give the speed-ups achieved relative to the time needed by a full pipeline (one core per pipeline stage) in all tests performed. We performed our tests with all combinations of configurations and arrangement strategies described earlier.

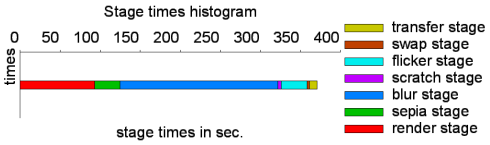

Figure 8. Overall stage running time using one SCC core.

The tests were made for the following three scenarios:

1) Unordered Arrangement: pipeline stages are assigned based on the processor order on the SCC. Leads to pipelines potentially starting/ending in the middle of the SCC grid.
2) Ordered Arrangement: pipeline stages are placed in parallel along the SCC grid to aid in communication flow.
3) Flipped-order Arrangement: pipelines are placed in the same manner as the Ordered Arrangement, however the direction of every other pipeline is reversed. This is done to obtain better memory controller usage.

In the first scenario, in which we use only a single core on the SCC for rendering, we gain an initial speed-up of 1.66 when using a full pipeline along the SCC as opposed to a single core. The relatively small speed-up results from the complexity of rendering, but also from the varying load within the pipeline. If we use two pipelines we achieve a speed-up of 1.94 (with respect to one pipeline) and a speed-up of 3.23 (with respect to only one core). Increasing the number of parallel pipelines further does not significantly increase the speed-up (see Figure 9). The maximal achieved speed-up is about 2.06 (w.r.t. one pipeline) and 3.44 (w.r.t. one core).

One striking fact is that the different pipeline arrangements on the SCC have no significant influence the speed-up of the system (see Figure 9). It seems that the rendering process is the major bottleneck of the system. If we remove

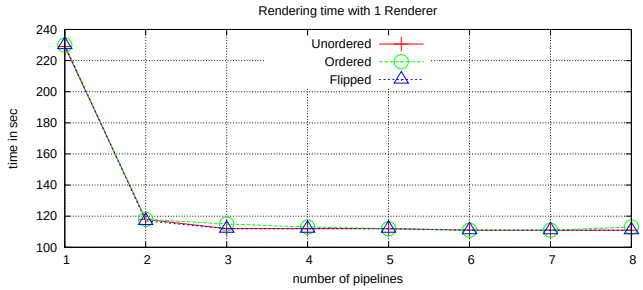

Figure 9. Processing time using one renderer with different numbers of pipelines. It is clear that this configuration does not scale well due to the rendering bottleneck.

all pipeline stages but the render and transfer stages, the walkthrough on one processor takes about 104 seconds and without the transfer stage it takes about 94 seconds. The rendering time alone is close to the best achieved walkthrough time in this configuration.

Due to the fact that, in general, the rendering task is the bottleneck within the pipeline, it is reasonable to distribute this task. In the second scenario, each pipeline is fed by a separate render stage that processes only a part of the complete image. Because of this, additional computation is necessary to adjust the viewing frustum of the camera. Again we tested this configuration with the different pipeline arrangements described.

In this configuration, if we only use one pipeline, the system achieved a slightly worse speed-up than in the previous configuration. The slightly smaller speed-up of 1.63 (w.r.t. one core) can be explained by the additional computations needed by this configuration. The extra computations that would be necessary for two or more renderers were not omitted when only using one renderer to give a true account of the speedups achieved. When two pipelines are used, a speed-up of 2.01 (w.r.t. one pipeline) and 3.26 (w.r.t. one core) is achieved. If we continue to increase the number of pipelines, we achieved further significant speed-ups, unlike the previous configuration. When three pipelines were used, a speed-up of 3.01 (w.r.t. one pipeline) and 4.83 (w.r.t. one core) is achieved. The speed-up continues to grow until we reach the maximum number of possible pipelines (7) in this configuration. Here the speed-up is 4.05 (w.r.t. one pipeline) and 6.89 (w.r.t. one core). The complete time for the walkthrough is about 58 seconds. Again, even in this configuration, the pipeline arrangement had no influence on the walkthrough time. For an overview of the performance in this configuration see Figure 10.

In our last scenario, we move the bottleneck of the render stage to the MCPC. Here the rendering program can access the memory exclusively without interfering with the other pipeline stages. However, we need an additional stage on the SCC that connects the MCPC with the different macro pipelines. This connector stage just receives the images from

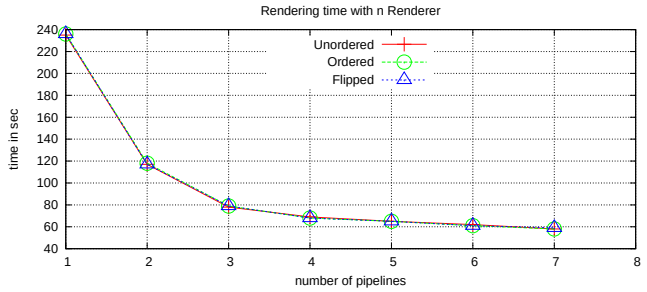

Figure 10. Processing time using one renderer per pipeline. The system scales better using this configuration.

the MCPC and distributes them onto the pipelines.

Like in the previous scenario, we do not get a remarkable speed-up if we use only one or two pipelines. The transfer of the images, the distribution by an additional pipeline stage, and the blur stage consume the saved rendering time. The connector stage also has to transfer its data from and to the memory, but it does not need the additional computations as in the previous scenario. It has to instead receive the images from the MCPC via UDP, which introduces additional overhead. Just as if a renderer stage is placed on the SCC, this stage causes network contentions during the data transfers with the blur stage. However, if we increase the number of parallel pipelines to three we gain a speed-up of 3.28 (w.r.t. one pipeline) and 5.46 (w.r.t. one core), which is similar to the previous scenario. If we use five pipelines, we achieve the maximal speed-up of 4.57 (w.r.t. one pipeline) and 7.49 (w.r.t. one core). Here the time needed for the complete walkthrough is about 51 seconds. The reduced memory accesses on the SCC influence the performance of the system positively. If we increase the number of pipelines further, we start to see a dip in performance. As seen in the previous two configurations, the arrangements of the pipeline stages on the SCC had no influence on the performance. The utilization of the different memory controllers also had no influence on the performance. For an overview of the performance in this configuration see Figure 11.

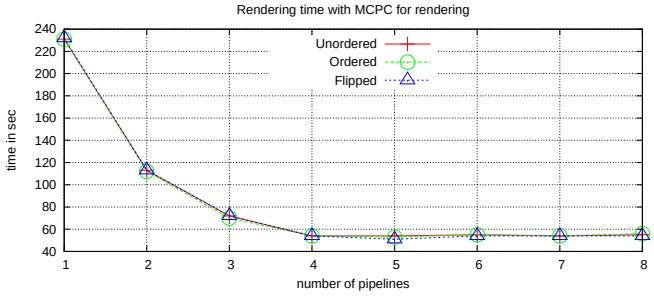

Figure 11. If the MCPC is used for rendering the system scales well until more than four pipelines are used.

Quite surprisingly, the arrangements of the stages on the SCC had no performance impact in all of our configurations. This is expected to be due to the lack of local memory for each core on the SCC. Unlike the Cell processor, the SCC only offers local caches. This means that, on the SCC, when a core wants to send a message (work) to its neighboring processor, the message actually has to travel first to the receiver processor's memory partition. The data must then be retrieved from memory by the receiver before it can begin its work. This happens at start of every computational stage on each processor. This situation does not occur on the Cell, for example, since the sending process can send data to the receiving process' local memory bank and the receiver does not need to look externally for this data, saving a significant amount of time. This lack of direct core-to-core communication hurts the overall performance achievable by the SCC. In this pipelining application, the data is processed once and then is not needed again by the processor. If the stages were to consist of much larger processes, then they would be able to take better advantage of these local caches. This might be true when the number of pipelines used meant that the tile size would fit directly into the local cache. In this situation, after paying for the initial memory fetch, everything could be processed locally. This presents a potential problem for the applicability of the SCC, as it seems it would only perform to its capacity for a small fraction of available programs. Generally speaking, local memory for each core might make the SCC platform much more powerful and flexible, allowing the programmer to dictate exactly how memory is managed to make full use of all available resources.

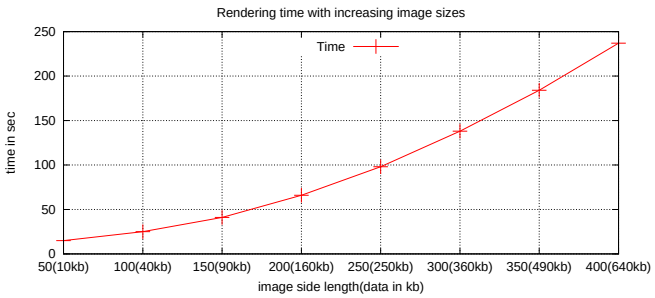

Figure 12. The amount of data has no significant impact to the processing time in each stage. There are jumps in the execution time caused by a better usage of the caches.

To analyze these effects in more detail we tested whether the amount of data processed in each stage has an impact on performance. Each core has its private 256 kib cache but no local memory. If the utilization of the cores' caches can help to overcome absence of local memory banks, suitable data packing might result in an acceleration. Hence we rendered images of different sizes on the MCPC and sent them through a single pipeline on the SCC. Figure 12 shows the resulting rendering times.

Surprisingly, there is no significant jump in the overall rendering time if the cores' cache size is exceeded. The

graph is slightly curved due to the overhead of transfering the images between the MCPC and the SCC. Due to the size of the send and receive buffers, the images cannot be sent in as a single message. The images must be divided into multiple sub-images and sent one after another.

For a comparsion, we performed a test consisting of each of the configurations described earlier on the *Mogon* HPC Cluster at Johannes Gutenberg-University Mainz, Germany. Here, each cluster node is equipped with 64 2.1 GHz cores. Thus, the cores' clock speed is roughly 3.94 times higher than the clock speed of the SCC's cores (533 Mhz). The results of these tests can be seen in Figure 13. Similar to the configurations on the SCC system, the images must be transferred via a network to the viewer which is placed on a differnt node. Similar to previous tests, in one configuration the renderer is also placed on an external node.

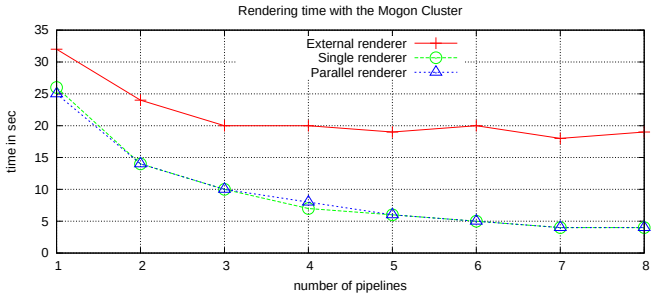

Figure 13. When a HPC with modern cores is used the rendering system is significantly faster.

Using the Mogon cluster, the rendering can be done at least three times faster than on the MCPC-SCC combination (which was the fastest on the SCC). The other configurations that were the slowest on the SCC system achieve the best performance on the cluster nodes. Using seven pipelines, the cluster is 13.5 times faster than the SCC system. All results are presented in Table I.

	1 pl.	2 pl.	3 pl.	4 pl.	5 pl.	6 pl.	7 pl.
1 rend., unordered	207s	107s	102s	102s	102s	101s	101s
1 rend., ordered	208s	108s	104s	103s	102s	101s	101s
1 rend., flipped	208s	107s	102s	102s	102s	101s	101s
n rend., unordered	235s	117s	78s	69s	65s	62s	58s
n rend., ordered	236s	118s	79s	68s	65s	61s	58s
n rend., flipped	236s	117s	79s	68s	65s	61s	59s
MCPC, unordered	231s	113s	72s	54s	54s	55s	54s
MCPC, ordered	231s	112s	70s	54s	53s	55s	54s
MCPC, flipped	232s	113s	72s	54s	51s	54s	54s
HPC, external rend.	32s	24s	20s	20s	19s	20s	18s
HPC, single rend.	26s	14s	10s	7s	6s	5s	4s
HPC, parallel rend.	25s	14s	10s	8s	6s	5s	4s

Table I
OVERVIEW OF THE RESULTS.

B. Power Consumption

Increasing the number of pipelines on the SCC naturally caused an increase in power consumption. This consumption increases linearly with the number of pipelines used. As before, the arrangement of the pipeline stages on the SCC cores had no influence on the overall power consumption. Figure 14 shows the power consumption for the configuration where the MCPC was used to render the images. The measurements for the other configurations look very similar.

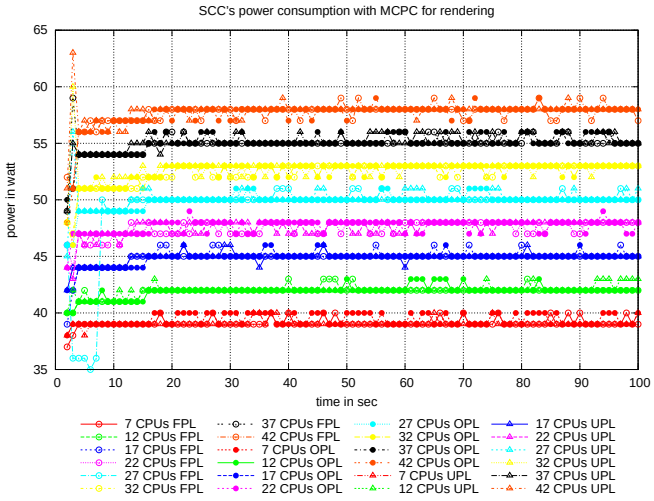

Figure 14. SCC power consumption increases linearly with the number of used pipelines. The flipped (FPL), the ordered (OPL), and the unordered (UPL) processor arrangements have no influence on the power consumed.

In the configuration where the MCPC was used as the renderer, we achieved the best speed-up if we use five macro pipelines. Here the SCC consumed only 50 watts compared to 22 watts while idling. The rendering of all images took only about 3.3 seconds, which means that the Xeon processor within the MCPC idles most of the time. During this time the consumed power increased from 52 watts (in idle mode) to 80 watts (during rendering). In the configuration which utilized one renderer per pipeline, we had to use the maximal possible number of pipelines to achieve the best result. Here the SCC consumed about 58 watts the entire time. In the heterogeneous system we consumed $3.3sec \cdot 28W + 51sec \cdot 50W = 2642J$ while the n-Renderer system consumes $58sec \cdot 58W = 3364J$. Thus, it is reasonable to use the hybrid MCPC and SCC approach in long running applications for a better performance/power consumption ratio.

C. Idle time

One of the major inefficiencies within the pipelines on the SCC seems to be the time that the cores must wait until they receive the next image to process. We evaluated the idle times for the configuration where the MCPC was responsible for the rendering. Figure 15 shows the idle times within the different stages when we use seven parallel pipelines. The plots in the diagram show the median and the quartiles of the time wasted on waiting on the next

input tile. In this scenario, the quartiles are very close to the median. It seems that the variances of the task times are small. The blur stage waits only for about 58 ms while the scratch stage waits for about 133 ms, if we use seven parallel pipelines. Accumulated over 400 frames, the blur stage waits for 23 seconds and the scratch stage waits for about 53 ms. The measurements show that the run time of the system is strongly influenced by these waiting times.

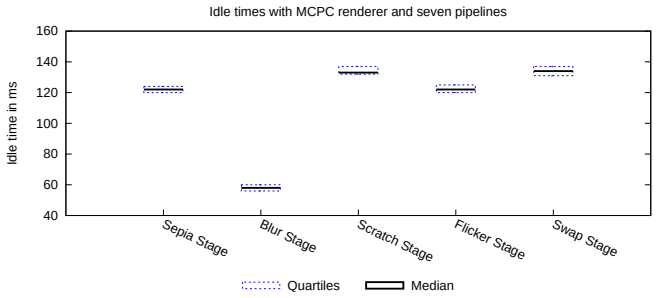

Figure 15. The wasted time while waiting to receive data from the previous pipeline stage.

D. Increasing the speed of the blur stage

The SCC provides the feature to manipulate the frequency and the consumed power of different parts on the processor. Figure 8 shows that the blur stage is the most time consuming filtering stage. An acceleration of the computations within this stage alone could result in a significant acceleration of the overall system. We changed the frequency of the cores processing this stage from 533 MHz to 800 MHz while the other cores used the default frequency of 533 MHz. To be able to increase the frequency of a given core, its voltage must also be increased from 1.1 V to 1.3 V. For this test we used a single macro pipeline with the rendering done on the MCPC.

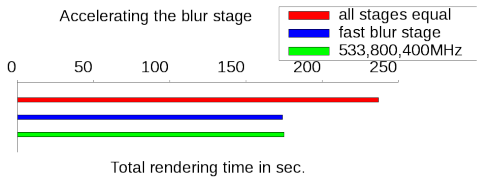

Figure 16. Increasing the frequency of the core computing the blur stage improves the overall performance significantly.

If the frequency of the core that executes the blur stage's computations is increased, it results in a significant acceleration. The total execution time of our test application is reduced from 236 seconds to 174 seconds (see Figure 16).

However, since increasing the frequency of a core increases the voltage, this results in higher total power consumption for the overall computation. Figure 17 shows this

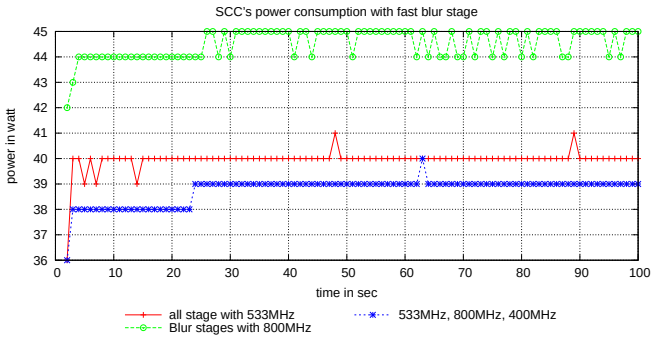

Figure 17. Improving macro pipeline's overall performance by increasing the blur stage's frequency results in significant higher power consumption.

increased consumption for the first 100 seconds of the application's execution.

The SCC only allows manipulating the voltage of multiple cores at once. Increasing the voltage for a specific core is not possible without increasing the voltage for all cores within a tile (see Figure 18). Thus, in this scenereo, more cores consume a higher amount of energy than necessary. For improved pipelining performance $4-5$ additional watts are required. This corresponds roughly to an increased consumption of about 10% while the overall performance improvement is about 36% relative to the configuration where all cores run at 533 MHz.

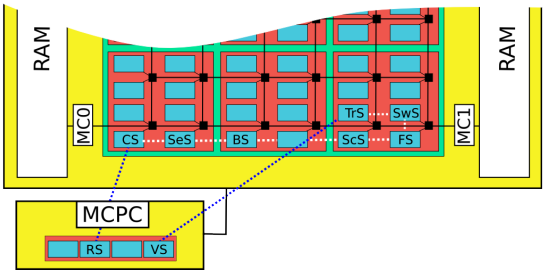

Figure 18. To increase only the frequency of the blur stage it must be placed in a separated tile.

To compensate for the additional power consumption, we decrease the frequency of the stages after the blur stage to 400 MHz. By doing this, we are decreasing the power usage of the cores computing the scratch, flicker, swap, and transfer stages to 0.7 V. These changes reduce SCC's average total power consumption to about 39 watts (see Figure 17). This is approximately 1 watt less than if all cores used a frequency of 533 MHz. In spite of this, the performance of this configuration is similar to the performance when only blur stage's frequency is increased (see Figure 16). In the configuration where only the core computing the blur stage is accelerated, the complete walkthrough's duration is 174 seconds. In the configuration where the frequency of the cores (following the blur stage) is decreased to 400 MHz the complete walkthrough take about 175 seconds. From

this we can conclude that signifcant returns can be made by adjusting the frequecies of the individual cores on the SCC.

VII. Conclusion

Overall, we have shown that the SCC and MCPC can be used in conjunction to produce a heterogeneous system capable of performing parallel macro pipelines. We believe users could expect similar performance to our image processing system for different pipelining applications. In contrast to other systems, the SCC allows one to build a larger number of parallel pipelines which share the total work.

One of the biggest disadvantages of this processor is the lack of a small, local memory for the individual cores. Because of this, all data accesses must be made over the grid network that connects the cores, via four memory controllers, with their memory. Small local and manageable memory banks per node would be a nice way to reduce the traffic on SCC's grid network. Storing the data locally would help to reduce the load on the network and could improve the SCC's applicability for parallel macro pipelining.

Finally, we showed that the SCC allows saving energy and increasing its overall performance if we configure the performance of the cores related to their specific tasks. We think this feature is the most promising one. Adjusting the cores' frequency and power consumption saves energy and produces less heat, which is an important issue especially for HPC system providers. Reducing the energy required for computation would result in a reduction of the costs to run their systems.

Acknowledgment

The authors would like to thank the Intel company for the provision of a SCC developer kit. This work was supported by the German Ministry for Education and Research (BMBF) under project grant 01|H11004 (ENHANCE).

References

[1] Intel Labs, *The SCC Platform Overview, Revision 0.7*, 2010.

[2] ——, *SCC External Architecture Specification (EAS), Revision 1.1*, 2010.

[3] ——, *The SCC Programmer's Guide, Revision 1.0*, 2010.

[4] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," in *ACM SIGGRAPH ASIA 2008 courses*, ser. SIGGRAPH Asia '08, 2008, pp. 35:1–35:11.

[5] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[6] R. David, D. Chillet, S. Pillement, and O. Sentieys, "DART: A Dynamically Reconfigurable Architecture Dealing with Future Mobile Telecommunications Constraints," *Parallel and Distributed Processing Symposium, International*, vol. 2, p. 0156, 2002.

[7] N. Tabrizi, N. Bagherzadeh, A. H. Kamalizad, and H. Du, "Mars: a macro-pipelined reconfigurable system," in *In CF '04: Proceedings of*, 2004, pp. 343–349.

[8] G. Ghare and S.-Y. Lee, "Dynamic Reconfiguration of a PMMLA for High-Throughput Applications," in *12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP 98)*. IEEE Computer Society, 1998, pp. 1–6.

[9] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008.

[10] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[11] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] Advanced Micro Devices, Inc., *ATI™ CrossFire™ Pro - User Guide*, 2008.

[13] Nvidia, *SLI Best Practices - 02/15/2011*, 2011.

[14] L. Logix, *Graphics task distribution performance scaling without compromise*, September 2010, technical report, whitepaper.

[15] T. Süß, C. Jähn, and M. Fischer, "Asynchronous Parallel Reliefboard Computation for Scene Object Approximation," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. Eurographics Association, May 2010, pp. 43–51.

[16] H. Xiong, H. Peng, A. Qin, and J. Shi, "Parallel occlusion culling on GPUs cluster," in *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, ser. VRCIA '06, 2006, pp. 19–26.

[17] T. Schumacher, T. Süß, C. Plessl, and M. Platzner, "FPGA Acceleration of Communication-Bound Streaming Applications: Architecture Modeling and a 3D Image Compositing Case Study," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–11, 2011.

[18] R. Rotta, J. Traue, and N. J. Prescher, Thomas, "Interrupt-Driven Cross-Core Invocation Mechanisms on the Intel SCC," in *Proceedings of the Many-core Applications Research Community Symposium MARC@RWTH Aachen University*, 2012, pp. 1–6.

[19] P. Reble, C. Clauss, M. Riepen, S. Lankes, and T. Bemmerl, "Connecting the Cloud: Transparent and Flexible Communication for a Cluster of Intel SCCs," in *Proceedings of the Many-core Applications Research Community Symposium MARC@RWTH Aachen University*, 2012, pp. 13–19.

[20] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "A Coherent and Managed Runtime for ML on the SCC ," in *Proceedings of the Many-core Applications Research Community Symposium MARC@RWTH Aachen University*, 2012, pp. 20–25.

[21] K. Pallister, *Game Programming Gems 5 (Game Programming Gems)*. Rockland, MA, USA: Charles River Media, Inc., 2005.