Programming hybrid systems with implicit memory based synchronization

Jens Breitbart
Research Group Programming Languages / Methodologies
Universität Kassel, Germany
Email:jbreitbart@uni-kassel.de

***Abstract*—In the last years CPU performance increases came with an increase in software development complexity. One of the next big changes in CPU architecture may be so-called hybrid multicore chips, which combine both multicore and manycore technologies on the same chip. Unfortunately, this increase in performance again may lead to an increase in development complexity. In this paper we suggest a dataflow driven programming model for hybrid multicore CPUs able to decrease complexity. We designed our model to work efficiently on non cache-coherent systems and in NUMA scenarios, which may both be issues exposed from the hardware in the near future. Our model is based on single assignment memory combined with PGAS-like explicit placement of data. Reading uninitialized memory results in the reading thread to be blocked until data is made available by another thread. That way threads can easily work in parallel and fine grained synchronization is done implicitly. Single assignment memory removes any cache coherency issues, whereas explicit data placement easily copes with NUMA effects. We implemented our model in a proof of concept C++ library for CUDA and x86 architectures, and use the scan algorithm to demonstrate that it is fairly easy to coordinate two GPUs and multiple CPU cores.**

I. Introduction

In the last decade we saw a large increase in single thread performance being made available transparently to developers. When this trend came to an end, vendors started putting multiple cores in one CPU to still be able to increase performance of the whole chip, yet this performance increase came with increased complexity for developers. Lately performance increase using this technique seems to slow down, but manycore chips such as GPUs on the other hand continue to increase performance at a high rate and provide multiple times the performance of modern CPUs. Unfortunately, not all problems are feasible for manycore chips, and the rather slow connection between multicore CPUs and manycore GPUs makes it hard to effectively have both types of cores closely cooperate on solving tasks.

A possible solution for future hardware are so-called hybrid multicore chips, which combine different kind of cores on the same chip. This allows to increase overall chip performance and eases the communication between different kind of cores. Lately AMD started shipping its CPU and GPU hybrid chip named Llano, which is currently the only hybrid chip in mainstream market. Using both components together again comes with an increased complexity for developers. We suggest a new programming model to reduce the complexity, also considering other upcoming hardware challenges:

- Hybrid CPUs consist of different kind of cores, e. g. throughput and latency optimized cores. Having such components working together requires fine-grained pair-wise synchronization.
- It is unclear if future hybrid CPUs will be cache coherent.
- With chip size increasing and multiple memory controllers being added to one chip, NUMA effects may appear between the different kind of cores.

We designed our programming model to coordinate different kind of cores and to effectively tackle the above issues. Our model can be used to coordinate cores of the same kind, but specific models may be more effecient.

In our model, memory shared by different kind of cores is single assignment and thereby can have two states: it can contain data, or be uninitialized. A thread reading uninitialized memory is blocked until another thread writes to that memory. A thread writing data automatically wakes up all threads waiting for the written data. In the extreme one can use this synchronization on bit level, but bitwise synchronization is in general unnecessary so we apply this mechanism on batches of data. Oversaturating the system with threads allows to hide latency introduced by synchronization. To cope with NUMA effects we use explicit placement of data, so it can be controlled which cores can access the data more efficiently.

We expect this form of synchronization to be easy to use and debug, as one cannot have race conditions. The worst case is a deadlock of a thread reading data that is never written, however debugging such a case is mostly easier than identifying race conditions, as the threads are blocked, clearly naming which data dependencies are not satisfied. Single assignment data obviously eliminates the need for cache coherency, as data cannot be changed. Offering direct placement of data to cope with NUMA issues is expected to be easier to use than most implicit placement techniques when NUMA costs are high, but using implicit placement can be used in case NUMA issues are not important. Overall, the requirements for implementing our model are rather minimal. Our software implementation uses flags to indicate the state of single assignment data. The implementation does not require atomic operations, but only memory fences. The model also allows for more efficient hardware implementation, as e. g. waiting for data currently unavailable is similar to modern GPUs suspending threads until the requested data is made available. Crays XMT [1] architecture offers a similar synchronization technique to our in hardware. Previous dataflow architectures are based on similar concepts too, e. g. the Manchester Dataflow architecture [2] used memory addresses to synchronize single instructions.

We implemented our model as a proof of concept C++ library offering vector and matrix data structures i.e., communication partners may synchronize on tiles of these structures. The library currently does not directly target hybrid chips, but we use CUDA's unified virtual address space to simulate a hybrid architecture. Our test system consists of two Fermi based GPUs and an Intel Nehalem based Core i7. Due to the lack of a hybrid CPU we have not yet optimized our library, but demonstrate its functionality by using memory placement and fine granular synchronization to keep all two GPUs and up to four CPU cores busy. We implemented a variation of the well-known scan algorithm of Blelloch [3] modified for GPUs by Harris et al. [4] as an example. We modified the algorithm so it uses multiple GPUs and CPU cores. Due to the relative slow memory transfers between CPU and GPU, we can obviously not achieve speedups, but show that one can synchronize all devices with hardly any additional work for developers.

The paper is organized as follows. First, Sect. II discusses possible hybrid multicore CPUs and upcoming hardware challenges, followed by an introduction to our model in Sect. III. Section IV discusses data structure creation and usage based on our model. A discussion on software implication of our model is in Sect. V. Section VI gives an overview of the hardware we used for testing, whereas Sect. VII discusses our model implementation. The scan algorithm is discussed in Sect. VIII. The paper finishes with an overview of related work and conclusions, in Sects. IX and X, respectively.

II. Hybrid multicore hardware

Current shared memory hardware architectures are dominated by two approaches: latency oriented cache coherent multicore CPUs and throughput oriented manycore GPUs. As said before, multicore architectures seem to reach a point at which further performance increase becomes rather complicated [5], [6]. So maintaining cache coherency for all cores gets harder with each core. Manycore architectures continue to increase performance and currently a GPU provide multiple times the performance of a multicore CPU, yet not all problems are feasible for manycore architectures. In most current, systems CPUs and GPUs can only communicate using the rather slow PCI Express bus.

Future processors can be built as so-called hybrid multicore CPUs, which consist of different kind of cores each optimal for certain types of tasks. In practice there are currently hardly any such systems on the market, however we give a brief overview next and discuss the possibilities and problems of such hardware in the rest of this section. We refer to hybrid multicore CPUs as hybrid chips and call each unit that consists of core of the same type a module, so a hybrid chip consists of multiple modules each consisting of uniform cores.

Probably the most well-known hybrid chip of the last years was the Cell Broadband Engine [7], which was able to achieve high performance compared to other CPUs of its time. Future development of the Cell B.E. seems to be discontinued. A new hybrid chip is AMD's already mentioned Llano chip [8], which combines a quad-core CPU with a manycore GPU. The chip is focused on mainstream desktop use and not on high performance computing. Both modules share the memory interface, however not the same address space so data is not automatically shared [9]. We expect this limitation to be lifted with future hardware generations, but do not expect that current shared memory programming techniques are a good match for such systems due to the issues discussed next. Liu et al. [10] came to a similar finding, when evaluating programming models for Intel's upcoming MIC architecture, previously known as Larrabee. MIC consists of multiple x86 based in-order CPUs with large vector units. We identified three major challenges crucial for developing a programming system for upcoming CPUs.

No cache coherence As said, we expect hybrid chips to offer shared memory for all cores. We furthermore assume future hybrid chips to contain caches, but not necessarily that they will be coherent. In case hybrid chips will contain coherent caches our model is still correct. On non cache coherent systems, memory barriers for the whole CPU will be getting rather expensive. Communication between the different kind of cores should be minimized and well defined. We design our programming model to effectively cope with the restriction of non-cache coherent hardware.

Efficient fine grained synchronization Hybrid systems are most efficiently used when all cores are busy. There may be external factors like memory bandwidth, power consumption, or heat dispersion which can prevent a chip from being able to keep all its cores busy, yet optimally all cores are supplied with tasks matching their architecture. We design our programming model to be able to allow effective usage of all cores. One of the main issues arising from a hybrid chip is that there can be both latency (CPU like) and throughput (GPU like) oriented cores on the same chip for which a form of fine grained pair wise synchronization must be employed. For example, if a latency core must wait until all throughput cores have completed their tasks, it will result in poor load balancing for the latency core.

NUMA Another issue with current systems is that memory bandwidth is increasing rather slowly compared to processing power. A solution to increase memory bandwidth is using multiple memory controllers on the same chip, as it is e. g. done by Intel's SCC [11], which is a manycore architecture without a shared address space. Having the memory controllers at the different sides of the chip may lead to NUMA issues, that is accesses to a certain memory location have a higher latency for some cores. We take this issue into account for our programming model as well. For simplicity we expect there to be one memory controller per module for the rest of this work, however the overall observations stay true even if the practical setup is changed.

III. Programming model

In this section we first discuss the impact of the issues described in the last section on current shared memory programming models using OpenMP as an example. We furthermore discuss CUDA and OpenCL and their restrictions regarding hybrid systems. After that we detail our model and discuss how it efficiently solves the issues.

The OpenMP memory model defines the so-called `flush` as a operation at which all writes of the calling thread must be made available to all threads and all previously done writes by other threads must be visible to the calling thread. On non cache coherent systems this requires writing all changed data to memory and reread all data. A flush is automatically included in almost all OpenMP constructs. After a synchronization with e. g. a barrier all memory reads must provide the threads with up to date values, which can increase the cost of a barrier in a non cache coherent system. Both these issues make OpenMP not a good match for non cache coherent systems. OpenMP also offers lock-like mechanisms for synchronization, which impose rather high overhead and may therefore not be feasible for fine grained synchronization, as well. OpenMP locks require all writing and reading threads to set and unset the lock, which could easily result in contention on the lock even if data is not changed. OpenMP by itself does currently not support NUMA optimization.

NVIDIA's CUDA [12] and OpenCL [13] in general are rather similar regarding their programming model. They were originally designed for distributed memory between the different modules, and memory is only consistent after a barrier synchronization of two modules. It is possible to manually synchronize at a finer scale, yet this is rather complicated as it requires explicit memory barriers and atomic operations. Data from a different memory space is normally read by making an explicit copy into memory local to that module and developers must make sure that the data is up to date whenever they access it.

Our model defines how communication and synchronization between the different modules is handled. It is of course possible to use the same techniques within a module, however module specific models can be more effective. For example one can use CUDA to program throughput cores and use our model to synchronize with latency cores. Our model has two major aspects:

- Synchronization is defined in a dataflow-like form, that is we tie synchronization to reading/writing single assignment memory.
- PGAS-like explicit memory placement is used to deal with NUMA issues.

The model is based on memory that is shared by all cores. We expect memory to be single assignment only, so exactly one thread can write once to a specific memory location, whereas it can be read multiple times by all threads. As memory is single assignment, we can define two states for any memory location:

- memory is uninitialized or
- it contains data.

Synchronization is based on these states following two rules:

- If a thread tries to read from uninitialized memory, it is blocked.
- A thread writing data will unblock all threads waiting for the written data.

Dataflow-based synchronization mechanisms are currently hardly used in HPC programing, but have been discussed before. A detailed overview can be found in the related work section. We could use our form of synchronization on bit level, which however is not feasible for most hardware and unnecessary for most problems. We therefore define the *synchronization size* as the size at which synchronization is done. We call this concept *synchronization granularity* and the block of data to which synchronization is applied a *synchronization unit*. For example, consider an algorithm using tiled matrices. The synchronization size can be identical to the tile size by which a tile becomes a synchronization unit. As a result, threads are only able to read data of a completely written tile. Working on a form of data blocks is essential for most hardware architectures to achieve high performance, as it e. g. allows effective use of CPU caches or GPU scratch pad memory. Requiring some form of blocking for synchronization therefor hardly increases complexity as existing blocking schemes can be reused. The synchronization size can differ for different memory locations and the optimal synchronization size depends on the algorithm and hardware used. The model gives the opportunity to hide possible latency introduced by synchronization with oversaturating threads, provided that threads can be effectively suspended.

Since data is single assignment, we can effectively deal with non cache coherent systems, because there is obviously no coherency issue. Data can only be loaded after it has been written and cannot be changed after that. The only information to be kept coherent is the state of a memory location, yet this can be simply implemented by a flag per synchronization unit or even specific hardware support could be added. Previous dynamic dataflow architectures already had support for similar kind of status information in hardware using content-addressable memory, as it was e. g. done in the Goodyear STARAN [14].

Our model allows for fine grained pair-wise synchronization. The minimal synchronization size still depends on the hardware, but the overhead for the synchronization is rather low. Again a trivial implementation using an additional flag to store the state of the data only requires a memory barrier before the flag is set to indicate the data is available. When reading this memory the flag must be checked before accessing the data. Checking can spinwait or suspend the thread in case the data is not available. It is again possible to implement a more effective hardware solution for this kind of synchronization, as the problem by itself is similar to suspending a reading thread until its requested data is in the cache.

We use explicit memory placement of data to solve the NUMA issues discussed in the previous section. The form of placement is similar to e. g. the partitioned global address space (PGAS) model in which the place where data is stored is part of the type of the data. For example, if CPU like cores generate data that is read by throughput cores multiple times, it can be placed near the throughput cores to speed up reading the data. In case, the hardware does not include any NUMA issues, the placement of data can be ignored.

The model as discussed here cannot only be used to program hybrid multicore CPUs, but can also easily be extended to support clusters using a PGAS approach. Synchronization can be done identical to what we described before and even the data placement can be used to store data at certain nodes to speed up reading.

IV. Data structures

In this section, we describe how synchronization through matrix and vector data structures can be realized in our model. Afterwards, we will sketch examples of their usage. We have implemented both data structures, but not all distributions discussed below. Note that the model is not limited to these data structures or distributions.

A data structure in the model is extended by two attributes

- synchronization size and
- data distribution

Both attributes are important, yet their values are hard to determine as they are influenced by both hardware and algorithm used. As a result they should be changeable without changing the data structures itself. In general synchronization size should directly be exposed to developers, so they can choose the best matching value. Data distribution can be done in several ways and we discuss some examples next.

A simple data structure is a tile based matrix that stores all data in the same module. The matrix stores data in tiles, which is well known and used for, e. g. increased cache utilization. The synchronization size of the matrix is identical to the size of the tiles, whereas the tile size is a factor to be considered during algorithm design. The data of such a matrix can be accessed by all modules. The same approach is used for the vector data structure, except that data is split in stripes not tiles.

In case the memory bandwidth provided by one module is not sufficient, tiles can be distributed across all modules. Such a distribution would allow all memory controllers to be used when reading the matrix, which provides a higher overall bandwidth at the cost of having no single module with low latency access to the whole data.

A vector with a synchronization size of one can easily be used to sum up input values, while part of the input is still being calculated. This can be implemented by using a CPU thread just reading vector elements and locally sum up all values. The CPU thread is blocked whenever the input is not yet ready.

The matrix data structure can be used to have throughput cores use it in computation, while parts of the matrix are still being read from file by a latency core. Synchronization is done implicit. Data placement is not important regarding correctness of the program, but only regarding performance.

V. Software implications

Our model breaks with a set of assumptions common in most well used programming models, yet we believe that software development complexity is not increased. We first discuss the single assignment nature of our model related to current CUDA programming and afterwards discuss synchronization granularity and memory placement regarding current shared memory approaches.

GPUs require off-chip memory accesses to be minimized due to their high latency. In practice this results in GPU kernels storing interim results on the chip and only final results are stored in off-chip memory. It is uncommon to modify data written once to off-chip memory in the same kernel. The final results are read in other kernels or by the CPU. This form of communication is in no way limited by the single assignment approach, as data is not changed after it is written. A common exception for this is that memory is reused to prevent allocating new memory, yet memory can be freed and new memory can be allocated in our model offering a direct solution.

Using locks often use a form of synchronization granularity, even though not explicitly. In most scenarios it is unreasonable to set a lock for every modification of a single entry in a data structures, yet a form of batch processing effectively reduces the overhead. Furthermore it is common to try to work on batches of data for increased cache utilization. Our model explicitly incorporates this technique as synchronization granularity and even though it requires explicit notion of the synchronization size, the problem by itself is not new.

The probably most common memory placement technique in NUMA systems is first-touch by which a memory page is placed close to the node writing to it the first time, yet this behavior may not be beneficial for a hybrid multicore chip. Write instructions can often be implemented as a fire-and-forget instruction, whereas latency resulting from reading data can directly harm performance. If using first-touch a latency core will have a high latency to data written by its throughput cores, which is obviously not desirable. Furthermore, current placement techniques are implicit and hardly transparent. It is hard to keep track of where data is stored, since its location is not explicitly visible. Explicit placement of data offers a more direct approach, which forces developers to deal with memory placement and may thereby look more complex at first. However, in case NUMA issues are important, direct control can be more flexible and easier to maintain.

VI. Hardware setup

Due to the lack of a real hybrid multicore system we tested our model on an Intel Core i7 920 with both a NVIDIA GeForce GTX 480 and a NVIDIA Tesla C2050.

- Core i7 920 is a quadcore CPU based on Intel's Nehalem architecture running at 2.66 GHz.
- GeForce GTX 480 is a GPU based on NVIDIA's Fermi architecture and consists of 15 so-called multiprocessors each with 32 cores resulting in a total of 480 cores. The GPU has access to 1.5 GB global memory, which is high bandwidth memory directly put on the same board as the chip.
- Tesla C2050 is a GPU based on NVIDIA's Fermi architecture as well, but only consists of 14 multiprocessors resulting in a total of 448 cores. The GPU has access to 3 GB of global memory.

We use NVIDIA's CUDA to program the GPU and use its so-called *unified virtual address space* to simulate a hybrid multicore chip. By using the virtual address space, CPU main memory and the global memory of all GPUs are mapped in the same memory address space. In general each device can access every memory location.

- Both GPUs can access CPU main memory, if allocated as page locked memory, also known as pinned memory. Developers can allocate memory to be cache coherent with the CPU, or allocate memory as write combined, which prevents the CPU from caching it. Accessing write combined memory can result in a performance increase of up to 40% [12] for the GPU at the cost of increased CPU access latency.
- The CPU can access GPU memory by explicitly making a partial copy of GPU memory into main memory using a CUDA specific `memcpy` function.
- If both GPUs are Tesla cards they can directly read and write each others global memory, however this is not possible in our system. This is a pure hardware limitation and not a limitation in our model or our implementation.

Even though there is a shared address space, there are no atomic operations for all devices, but atomic operations are atomic for one device only.

Memory accesses from a GPU to main memory or from a CPU to global memory are DMA transfers over the PCI Express bus, which has a rather high latency and a maximum bandwidth of 8 GB/s. Remote memory accesses are thereby more expensive than for hybrid multicore systems. This problem makes it hard to judge the performance of our implementation regarding hybrid chips, so we have left most performance optimizations of our implementation out for future work.

VII. Model implementation

We have implemented the model in a set of data structures available as a C++ library. The library has been tested with CUDA and x86 architectures and is available open source[1]. The implementation was done together with the PGAS implementation of the model, but at the time of this writing not all features discussed here will work together with their PGAS counterparts. The PGAS and hybrid part share the implementation for synchronization and data distribution. Furthermore, e. g. CPUs in a remote node can access local pinned memory, however this is currently not true for remote GPUs, and local CPUs cannot access remote global memory. Future work will unify both implementations. A detailed discussion of the PGAS implementation is out of scope for this work.

As the current target architectures do not have direct support for our synchronization mechanism, we implemented it by adding a flag, as discussed before. The flag is stored directly in front of a synchronization unit. We put a memory barrier before setting the flag to make sure all data is written before another thread can read the flag as available. The flag is set using volatile variables, so the flag itself is directly made available. The flag must not be set atomically, as in a correct program it is not changed by multiple threads, however our implementation uses atomic operations to try to detect possible developer errors. In case two threads try to change the same flag, an assertion is triggered and the program is stopped. As our system does not provide atomic operations for all devices the checks cannot detect all errors and is only a best effort approach.

Our flag based implementation is rather lightweight. The costs for writing a synchronization unit are increased by the cost of setting the flag. There should be no contention on modifying the flag nor will there be any false sharing with

[1]https://github.com/jbreitbart/sa-pgas

reasonable large synchronization sizes. In case the flag is not yet set, reading a synchronization unit will spinloop until the flag is set. Note that only the reads in the loop must not be from cache on non cache-coherent systems, as in case the flag was once set it will not be changed. In case the flag is already set, the costs added are low.

We allow explicit data placement by using what we call distributions. Distributions must be supplied whenever a data structure is created and are used as both an allocator and random access iterator, that is the distribution allocates all data required for the data structures and manages all accesses to it. Data access is managed by three basic functions.

- `get()` manages accesses to the synchronization unit and blocks in case data is not yet written.
- `get_unitialized()` returns a pointer where data can be written to.
- `set()` marks the synchronization unit to be written.

Data writes are in general done in three steps.

1) `get_unitialized()` is called. The functions returns a pointer to a synchronization unit.
2) Data is written to the pointer returned in step 1.
3) `set()` is called marking the data as available.

The optimal distribution most likely depends on the specific task, so we made distributions easy to create. We supply low level allocators and pointer like random access iterators, which take care of memory allocation and synchronization, as well as data access itself. The allocators are specific for a memory space, and offer implementations for all three data access functions. When the low level constructs are used, a distribution must only implement a mapping of a linear address space to the allocated memory blocks.

As an example, a minimal distribution maps a linear address space to a single block of pinned memory, so all data is stored close to the CPU and accessible by CPUs and GPUs. A distribution could also allocate memory on both GPUs and distribute the data stripe-wise between the two memory spaces. The implementation of this distribution can allocate two memory blocks in its constructor and implement the data access functions to forward the accesses to the low level iterator responsible for the requested memory. The overhead of the distribution depends on how complex the mapping of the data is, in case of e. g. the trivial pinned memory distribution the compiler will most likely inline all function calls resulting in hardly any overhead.

VIII. Example Scan implementation

In this section we first introduce the CUDA parallel prefix sums implementation suggested by Harris et al. [4] and afterwards outline how we distributed the workload to the different devices.

All prefix sums – also known as scan – is an important parallel building block used in a wide variety of algorithms. Scan takes an input array $[x_0, x_1, ..., x_{n-1}]$ and a binary associative operator $+$ with the identity I as input and returns $[I, x_0, (x_0 + x_1), ..., (x_0 + x_1 + ...x_{n-2})]$. Harris et al. suggested a work-efficient implementation for CUDA, which is split in three steps. In the first step the whole array is subdivided into blocks and a local scan is performed on every block. In this step the block sums of every block are written to an auxiliary array (SUM). In step 2, SUM is scanned and in a third step, the result of the scan are used to update the original array, so it contains the final result. Figure 1 gives a basic overview of the algorithm in which we added the filling of the input array as a step 0.

Our implementation uses the CPU for filling the input array and the scan of the block sums (steps 0 and 2), whereas we use one GPU for the block wise scan and another one for producing the final results, which allows the systems to automatically form a pipeline. We use 5 vectors in our scan code, all storing the data in pinned main memory.

- The input array filled in step 0 uses a synchronization size of 256, which is a reasonable size for the block local scan on the GPU.
- The scanned blocks are stored in another vector again with a synchronization size of 256.
- The block sums are stored in a vector with a synchronization size of 1, so the CPU can continue to compute the scan as soon as possible.
- The scanned block sums are again stored with a synchronization size of 1.
- The final result vector uses a synchronization size of 256.

As soon as the CPU has generated 256 input elements, the first GPU multiprocessor can start with a block local scan. The CPU on the other hand can scan SUM as soon as the first block sum is available. After the first block sums are scanned, the second GPU can start computing the final result. Whereas this distribution of work does not provide high performance on the used system due to the slow communication link, it demonstrates that using multiple modules in parallel must hardly increase overall programming complexity. As a downside, the algorithm uses a rather high amount of memory. Previous dataflow architectures solved such issues by treating all but the final result as transient memory, which were automatically removed when no longer needed. Implementing such a feature in software on a hybrid system may turn out to be rather complex and future work is required to identify if e. g. a reference counting approach offers sufficient results. Experiments with reference counting in the PGAS model seem to provide a reasonable usability. The current implementation requires the CPU to free the memory when it is no longer used, e. g. in the scan algorithm the input data can be deleted as soon as the block sum scan is complete, because it will not be used again.

Scan is normally computed on preliminary results to

Problem size (in 256 element blocks)	50	100	200	400	500	1000
Runtime (ms)	5.833	1.1595	2.3537	4.6528	5.8126	11.6958
Input (ms)	5.806	1.1553	2.3468	4.6404	5.7974	11.6668
Pure computation (ms)	0.0027	0.0042	0.0069	0.0124	0.0152	0.029

Table I
RUNTIME FOR A SINGLE SCAN COMPUTATION WITH DIFFERENT INPUT SIZES.

achieve the final result. To simulate such a computation we slowed down the generation of input by doing 50,000 additions per 256 elements. We measured performance of our implementation with several input sizes (Tab. I). The table shows:

- ***runtime*** The total runtime including the generation of the input data.
- ***input*** The time required to generate the input data.
- ***pure computation*** The time required to complete the computation after the input was generated.

We can see that time required for input generation increases with larger problem size, but pure computation time is constant. This gives a clear indication that the pipeline works as expected and almost all computations are done while the input data is generated.

IX. RELATED WORK

Overall, the topic of this paper is related to a large spectrum of research. Dataflow hardware has been an active area of research in the 1970s and 1980s, as exemplified by the Manchester dataflow architecture [2] or the Goodyear STARAN [14]. From a current viewpoint these approaches were unable to scale effectively and were replaced by other architectures. However, the techniques partially returned in e. g. out of order architectures, although at a smaller scale than originally planned. Single assignment memory is not used in current mainstream languages, but of course used in functional programming. For example, SISAL (Streams and Iteration in a Single Assignment Language) [15] is a functional programming language using implicit parallelism extracted from a dataflow graph, yet relies on compilers to generate dataflow graphs and targets SMT systems. Haskell uses so-called MVars [16], which are atomically filled communication channels between threads that block the reading thread in a similar fashion to our single assignment memory. Cray's Chapel [17] offers synchronization variables, which can work similar to an MVar or to our single assignment memory, yet they are only supported for basic data types and do not support synchronization granularity. Chapel's memory model does not target hybrid systems and Chapel in general has not yet been used to program a hybrid multicore CPU. One of the first hardware architectures supporting synchronization bits was the Denelcor HEP [18], which marks values as unavailable after they have been read. After working on the HEP, Smith has been following the concept of tying synchronization to memory accesses for a longer time [19], yet the concept has not been used in a major HPC programming system. A current architecture supporting this mechanism is Cray's XMT [1].

X. CONCLUSION

In this paper, we suggested a programming model easing the transition from current multicores to hybrid systems for developers. Our model is based on single assignment memory and provides support for fine grained synchronization and non cache coherent systems, as well as the ability to deal with NUMA issues. We implemented a prototype of the model as a C++ library demonstrating its ease of use. The model by itself is not limited to hybrid systems and may turn out to be feasible even for clusters, yet research in this area is still ongoing. As most of the work till now has been done regarding the model, future work will improve the implementation especially focus on real hybrid systems and test the model on other problems.

REFERENCES

[1] Cray Inc., "Introducing the Cray XMT Supercomputer," Cray Inc., 2010.

[2] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Commun. ACM*, vol. 28, pp. 34–52, January 1985.

[3] G. E. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Trans. Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[4] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

[5] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling," in *Proceedings of the 32nd annual international Symposium on Computer Architecture*, 2005, pp. 408–419.

[6] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[7] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, pp. 559–572, 2007.

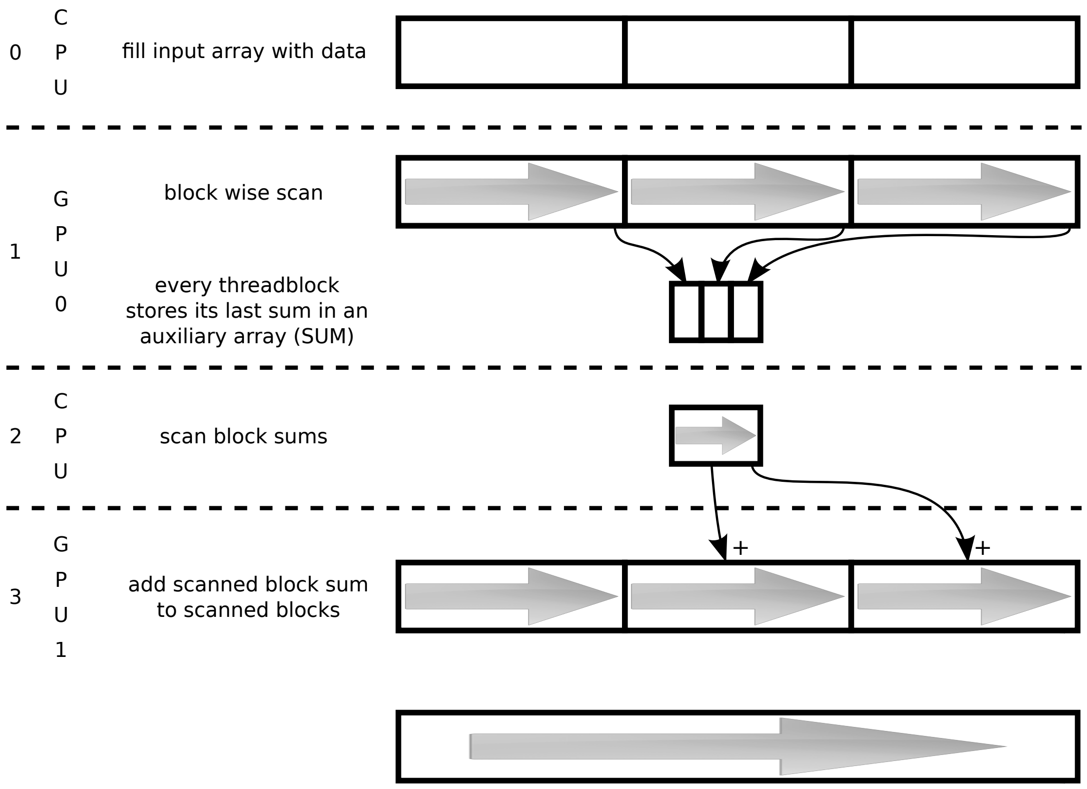
0
C
P
U
fill input array with data
1
G
P
U
0
block wise scan
every threadblock stores its last sum in an auxiliary array (SUM)
2
C
P
U
scan block sums
3
G
P
U
1
add scanned block sum to scanned blocks
+
+

Figure 1. Overview of the scan algorithm.

[8] AMD, "AMD Fusion Family of APUs," 2011.

[9] ——, "AMD Accelerated Parallel Processing OpenCL." AMD, 2011.

[10] W. Liu, B. Lewis, X. Zhou, H. Chen, Y. Gao, S. Yan, S. Luo, and B. Saha, "A balanced programming model for emerging heterogeneous. multicore systems," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, 2010.

[11] J. Held, ""'Single-chip cloud computer", an IA tera-scale research processor," in *Euro-Par 2010 Proceedings of the 2010 Conference on Parallel Processing*, 2011, pp. 85–85.

[12] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 4.0," NVIDIA Corporation, 2011.

[13] "The OpenCL Specification. Version 1, revision 43," May 2009.

[14] Goodyear Aerospace Cooperation, "STARAN APPLE Programming Manual," 1972.

[15] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A Report on the Sisal Language Project," *Journal of Parallel and Distributed Computing*, vol. 10, pp. 349–366, 1990.

[16] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent Haskell," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[17] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade High Productivity Language," in *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[18] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Society of Photo-Optical Instrumentation Engineers Conference Series*, 1981, pp. 241–+.

[19] G. Alverson, R. Alverson *et al.*, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in *Proceedings of the 6th International Conference on Supercomputing*, pp. 188–197.