Asymptotically Optimal Merging on ManyCore GPUs*

Arne KUTZNER†(a), Nonmember, Pok-Son KIM†(b), Member, and Won-Kwang PARK†(c), Nonmember

SUMMARY We propose a family of algorithms for efficiently merging on contemporary GPUs, so that each algorithm requires $O(m \log (\frac{m}{n} + 1))$ element comparisons, where $m$ and $n$ are the sizes of the input sequences with $m \leq n$. According to the lower bounds for merging all proposed algorithms are asymptotically optimal regarding the number of necessary comparisons. First we introduce a parallelly structured algorithm that splits a merging problem of size $2^i$ into $2^i$ subproblems of size $2^{i-1}$, for some arbitrary $i$ with $(0 \leq i \leq l)$. This algorithm represents a merger for $i = l$ but it is rather inefficient in this case. The efficiency is boosted by moving to a two stage approach where the splitting process stops at some predetermined level and transfers control to several parallelly operating block-mergers. We formally prove the asymptotic optimality of the splitting process and show that for symmetrical sized inputs our approach delivers up to 4 times faster runtimes than the thrust::merge function that is part of the Thrust library. For assessing the value of our merging technique in the context of sorting we construct and evaluate a MergeSort on top of it. In the context of our benchmarking the resulting MergeSort clearly outperforms the MergeSort implementation provided by the Thrust library as well as Cederman’s GPU optimized variant of QuickSort.

key words: parallel algorithms, GPGPU, complexity, merging, sorting

1. Introduction

Merging denotes the operation of rearranging the elements of two adjacent sorted sequences of size $m$ and $n$, so that the result forms one sorted sequence of $m + n$ elements. A merging algorithm is regarded as stable, if it preserves the initial ordering of elements with equal value. There are two significant lower bounds for merging. The lower bound for the number of assignments is $m + n$ because every element of the input sequences can change its position in the sorted output. As shown e.g. in Knuth [1] the lower bound for the number of comparisons is $\Omega(m \log (\frac{m}{n} + 1))$, where $m \leq n$.

GPGPU denotes a set of technologies for harnessing the massive power of modern multicore GPUs in the context of general purpose computing. An introduction into the architecture of modern GPUs as well as the special aspects of their programming is given e.g. in [2]. The primary problem in the context of GPGPU programming is the clever decomposition of the overall problem into subproblems, so that these can be solved independently by many parallelly executed threads. As severe restriction in the context of this process all threads have to work on the same code basis. Merging on the foundation of GPGPU was already discussed in [3], however only for symmetrically sized inputs.

The architecture of modern GPUs represents in a wider sense a PRAM model, where we have a parallel machine consisting of a collection of consecutively numbered processors that share a common global memory. So the research done for merging on PRAM models ([4]–[8]) has significance in the context of the research on merging on GPUs. However, aside from some basic resemblances we will go a way not followed in the literature so far. Our starting point will be a principle of parallel symmetric splittings that has its origins in a similar principle proposed in [9] in the context of minimum storage merging. Using repeated symmetric splittings we will decompose the given merging problem into equally sized subproblems. Such a decomposition can continue until each sequence consist merely of one element, but this is rather inefficient in praxis. Alternatively the decomposition can stop at some predetermined sequence size and the remaining merging can be performed by block-mergers as e.g. bitonic mergers. The even decomposition causes even loads on thread level and this in turn avoids idle threads. The significance of such an even decomposition was already mentioned by Francis and Mathieson in [8]. They propose a significantly less efficient algorithm that can compute segmentations identical to ours. Let $m, n$ ($m \leq n$) be the sizes of both input sequences and $p$ the number of available processor in the PRAM model. If we have $p = m + n$, i.e. each processor computes the final position of a single element, then their approach requires altogether $O((n + m) \log (m + 1))$ many comparisons for the construction of the sorted outcome. Our approach keeps asymptotic optimality and this allows maximally $O(m \log (\frac{m}{n} + 1))$ comparisons.

Our complexity analysis will be focused on counting the number of comparisons because of their significance in the context of merging. This unit of measurement can be disputed, because it does not express anything about the degree of parallelism. However, a notion like “runs in time $x$” is difficult to apply in the context of GPUs due to their behavior of implicit serialization in the context of branching on thread-block level. Further it is difficult to integrate a pa-
rater like “number of processors $p$” because the central parallel entities in the context of GPGPU are threads that have a many to one relationship regarding physical processors. In order to defend our decisions to count comparisons we would like to remark, that the benchmarking will show for all of our algorithms a run-time behavior that reflects to a remarkable degree the lower-bound for the number of comparisons.

We will start with an introducing example for explaining the basic idea of merging by symmetric splittings. Afterward we give a formal definition of a basic algorithm, prove its complexity regarding the number of comparisons and explain its implementation in a GPGPU context. By switching to a two-stage approach we will receive an extensible family of performant merging algorithms. After inspecting the runtime behavior of these algorithms we embed them into a GPU optimized variant of the MergeSort algorithm. We finish by reporting about benchmarking for this form of sorting.

2. The Split-Sequence Algorithm

We start with a brief introduction into our approach for merging. Let us assume that we have to merge two sequences $u = (0, 3, 5, 9)$ and $v = (1, 2, 7, 8)$ stored in some array $A$, where $u$ is in $A[0..3]$ and $v$ is in $A[4..7]$. Using the outcome of a sequence of symmetrically shaped comparisons we split $u$ into $u_0u_1$ and $v$ into $v_0v_1$, so that in the finally merged sequence the elements of $u_0v_0$ are followed by the elements of $u_1v_1$. In detail this splitting happens as follows: Starting from the leftmost position in $u$ and rightmost position in $v$ we symmetrically compare elements until we reach the pair of innermost elements of both input sequences. There can be at most one position, where the relation between the compared elements alters from ‘not greater’ to ‘greater’. This position indicates our split points. If we apply this technique, we will call it symmetric splitting, to our example, we get that the elements of $u_0 = \{0, 3\}$ and $v_0 = \{1, 2\}$ are followed by the pairs $u_1 = \{5\}$ and $v_1 = \{7\}$.

The outcome of this first step represents a decomposition of our merging problem for 8 elements into 2 equally sized merging problems for 4 elements. We will now continue by decomposing these 2 problems (Level 1 in Fig. 1) by repeatedly applying the technique described above. As result we get 4 problems of size 2 (Level 2 in Fig. 1). Finally, by applying the splitting scheme once again, we get 8 problems of size 1 (Level 3 in Fig. 1). The sequence of indices $(0, 4, 5, 1, 2, 6, 7, 3)$ that we have on Level 3 indicates the order of elements in the finally merged sequence. The process described above leaves the elements of both input sequences untouched. The actual reordering of elements has to be performed in an additional step on foundation of the sequence of indices received on Level 3. The symmetric splitting can be achieved efficiently on the foundation of a symmetrically progressing binary search.

Fig. 1 Example for repeated application of Split-Sequence.

For the sake of simplicity we constructed our introducing example so that we always operate on inputs of equal size. We will now generalize our splitting technique, so that it can cope with inputs of arbitrary form. Without loss of generality let us assume that the left input sequence $u$ is equal or shorter than the right one $v$. In this case we decompose $v$ into $b_1wb_2$, where $w$ is a centered section of $v$ with $|w| = |v|$, and apply our symmetric splitting on $u$ and $w$. We will now give a formal definition for our Split-Sequence algorithm.

2.1 Formal definition

Let $u$ and $v$ be two adjacent ascending sorted sequences. We define $u \leq v$ ($u < v$) iff $x \leq y$ ($x < y$) for all elements $x \in u$ and for all elements $y \in v$.

Without loss of generality we assume that the overall size of the input $u$ represents some power of 2, i.e. $|u| + |v| = 2^l$ for some $l \geq 0$. Inductively we will develop $l$ levels, so that each level $i$ ($0 \leq i < l$) consists of $2^i$ pairs of input segments originating from $u$ and $v$. In the following $(u_j^i, v_j^i)$ shall denote the $j$-th pair of blocks on level $i$, where $0 \leq j < 2^i$. $u_j^i$ or $v_j^i$ can be the empty sequence denoted by $\epsilon$.

We define $(u_0^0, v_0^0) = (u, v)$ and develop the input pairs for level $i + 1$ out of the pairs for level $i$ as follows:

If $|u_j^i| \leq |v_j^i|$ then

(a1) we decompose $v_j^i$ into $b_1wb_2$ such that $|w| = |u_j^i|$ and $|b_2| = |b_1|$. 

(a2) we decompose $u_j^i$ into $a_1a_2$ ($|a_1| \geq 0$, $|a_2| \geq 0$) and $w$ into $w_1w_2$ ($|w_1| \geq 0$, $|w_2| \geq 0$) such that $|a_1| = |w_2|$, $|a_2| = |w_1|$ and $a_1 \leq w_2$, $a_2 > w_1$.

(a3) we set $u_{j+1} = a_1$, $v_{j+1} = b_1w_1$, $u_{j+1} = a_2$ and $v_{j+1} = w_2b_2$.

otherwise

(b1) we decompose $u_j^i$ into $a_1wa_2$ such that $|w| = |v_j^i|$. 

Corollary 1. Let \(|u| + |v| = 2^t\) for some \(t > 0\). Then it holds
\[ |u_2^{t+1}| + |u_2^{t+1}| = |u_{2j+1}^{t+1}| + |v_{2j+1}^{t+1}| = 2^{t-1}. \]

Corollary 2. Let \(2^l (l > 0)\) be the size of our input \(w, v\). Then, on level \(0 \leq i \leq l\) \textsc{Split-Sequence} generates \(2^l\) pairs
\((u_0^i, v_0^i), (u_1^i, v_1^i), \ldots, (u_{2^l-1}^i, v_{2^l-1}^i)\) where it holds \(u_i^j + v_i^j = 2^i \) for all \(j = 0, 1, \ldots, 2^i - 1\).

During the symmetric splitting of \(u_i^j\) and \(w, v_i^j\) it always holds \(a_1 \leq w_2 \) and \(a_2 > w_1 (w_1 \leq b_2 \) and \(w_2 > b_1\) see Fig. 2). The treatment of pairwise equal elements as part of the outer blocks \((a_1, w_2)\) in (a2) and \((w_1, b_2)\) in (b2)) avoids the exchange of equal elements and so the reordering of these. Hence we get the following Corollary:

Corollary 3. \textsc{Split-Sequence} preserves the stability.

2.2 Worst Case Complexity

Without loss of generality we assume \(|u| \leq |v|\). Additionally, unless stated otherwise, let us denote \(m = |u|, n = |v| (m \leq n)\), \(k = \log m\) and \(m + n = 2^l, l \geq 0\). Further let \(m_l^i \) and \(n_l^i \) be \(|u|\) and \(|v|\) respectively. Initially (on the recursion level 0), it holds \(m_0^i = m \) and \(n_0^i = n\).

Now let \(\mathbb{N}\) be the set of nonnegative integers. In the following we introduce two definitions:

Definition 4. Let \(X^i \subseteq \{0, 1, 2, \ldots, 2^i - 1\}, i \in \mathbb{N}\) be a set consisting of indices \(j\) of \(m_l^i\) on recursion level \(i\). \(j \in X^i\) iff \(m_l^i \neq 0\).

Definition 5. The function \(\rho : \mathbb{N} \rightarrow \mathbb{N}\) is defined as follows:
\[ \rho(x) = \begin{cases} \lceil \log x \rceil + 1 & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases} \]

As already mentioned, on level \(i\), \(2^i\) pairs, i.e. \((u_0^i, v_0^i), (u_1^i, v_1^i), \ldots, (u_{2^i-1}^i, v_{2^i-1}^i)\), are generated. For generating input pairs for the next level \(i + 1\), each pair \((u_i^j, v_i^j)\) needs as many comparisons as \(\rho(\min(m_l^i, n_l^j)) \leq \rho(m_l^i) = \lceil \log m_l^i \rceil + 1 \text{ if } m_l^i \neq 0, n_l^j \neq 0\) and 0 otherwise. We can formally state and prove the following basic results:

Lemma 6. For each \(i, 0 \leq i \leq l\), \(|X^i| \leq \sum_{i=0}^{2^i-1} 2^i\) if \(0 \leq i \leq k\) and \(|X^i| \leq \frac{2^i}{m}\) otherwise.

Proof. Since each pair is divided into two pairs, it holds \(|X^i| \leq 2^i \leq m\) for all \(i, 0 \leq i \leq l\). Further, since the sum of all \(m_j^i\)s is always bounded by \(m\), it holds \(|X^i| \leq m\), for \(i > k\).

Lemma 7. Let \(x_1, x_2, \ldots, x_m\) be any nonnegative integers satisfying \(\sum_{i=1}^{m} x_i = m\). Then for each \(m\)-tuple \((x_1, x_2, \ldots, x_m)\), \(\sum_{i=1}^{m} \rho(x_i) \leq m\).

Proof. It holds trivially, since \(x \geq \rho(x)\) for each nonnegative integer \(x\).

Lemma 8. ([10] Lemma 3.1) If \( k = \sum_{j=1}^{2^l} k_j \) for any \( k_j > 0 \) and integer \( i \geq 0 \), then \( \sum_{j=1}^{2^l} \log k_j \leq 2^l \log(k/2) \).

Theorem 9. Split-Sequence is asymptotically optimal regarding the number of comparisons, i.e. it requires \( O(m \log(\frac{m}{n} + 1)) \) comparisons for merging two sequences \( u, v \) with \( |u| = m \) and \( |v| = n, m \leq n \).

Proof. We split our proof into two parts (see Fig.4), part I and part II. Part I covers recursion levels 0, 1, \( \cdots \), \( k = \lfloor \log m \rfloor \). Part II covers recursion levels \( k + 1, \cdots, l \), where \( l = \log(m + n) \).

Part I: The number of comparisons required on each recursion level \( i \), \( 0 \leq i \leq k \) is equal or less than \( \sum_{j=1}^{2^l} (\log(m_j^i) + 0.5) + 1 \). Since \( \sum_{j=1}^{2^l} (m_j^i + 0.5) \leq m + 2^{i-1} \leq (3m/2) \) and by Lemma 8, Split-Sequence performs at most \( 2^i + 2 \log((3m/2)/2^i) \) comparisons on each recursion level \( i \). So the overall number of comparisons for all levels is less than \( \sum_{i=0}^{k} 2^i (1 + \log(3m/2)) - i^2 \). Since \( \sum_{i=0}^{k} 2^i = (k-1)2^{k+1} + 2, \) Split-Sequence needs at most \( (1 + \log(3m/2))(2^{k+1} - 1) - (k-1)2^{k+1} - 2 \leq 2m(\log(3m/2)) - (k-2)2^{k+1} - 3 - \log(3m/2) \leq 2m(3 + \log(3/2)) - 3 - \log(3m/2) < 8m = O(m) \) comparisons for part I.

Part II: For each recursion level \( i = k + 1, \cdots, \log(m + n) \), the number of required comparisons does not exceed \( \sum_{j \in X} \rho(m_j^i) \). Further it holds for each \( i = k + 1, \cdots, \log(m + n) \) of \( \sum_{j \in X} m_j^i \leq m \) where \( |X|' \leq m \) (Lemma 6). Therefore, by Lemma 7, \( \sum_{j \in X} \rho(m_j^i) \leq m \). Thus the overall number of comparisons for all levels of part II is less than \( (\log(m+n) - \lfloor \log m \rfloor) + m = O(m \log(\frac{m}{n} + 1)) \) comparisons. □

3. Technical Realization and Pseudocode

Let us assume that our input \( w \) is stored in some array \( A \) and that \( (u',v') \) is a pair of segments with \( u' \) originating from \( u \) and \( v' \) originating from \( v \). If \( u' \) occurs in \( A[x:y] \) and \( v' \) occurs in \( A[z:w] \) we represent \( (u',v') \) by the single 4-tuple \( (x,y,z,w) \). According to the formal definition given before, we require \( 2^i \) 4-tuples for the representation of some level \( i \) during the split process. For the sake of simpleness we further assume that the size of \( A \) is some power of two.

Algorithm 1 Pseudocode of the Split and Map procedures (GPU-kernels)

**Split(A, Idx, i, seg_size)**
1. \( pos \leftarrow irott seg size \)
2. \((start_J, end_J, start_r, end_r) \leftarrow Idx[pos]\)
3. \(delta_left \leftarrow end_J - start_J - (seg_size/2)\)
4. if \( \text{delta_left} \geq 0 \)
5. then \( r \leftarrow start_J + (seg_size/2)\)
6. \( l \leftarrow r - end_J + start_J\)
7. \( delta \leftarrow l - end_J\)
8. else \( r \leftarrow end_J\)
9. \( l \leftarrow start_J\)
10. \( delta \leftarrow l + end_J + delta_{left} \)
11. while \( l < r \)
12. do \( m \leftarrow \lfloor (l + r)/2 \rfloor \)
13. if \( A[m] \leq A[delta - m - 1] \)
14. then \( l \leftarrow m + 1 \)
15. else \( r \leftarrow m \)
16. cut \leftarrow delta - l \)
17. \( Idx[pos] \leftarrow (start_J, l, start_r, cut)\)
18. \( Idx[pos + (seg_size/2)] \leftarrow (l, end_J, cut, end_r)\)

**Map(A', Idx, i)**
1. \((start_J, end_J, start_r, end_r) \leftarrow Idx[i]\)
2. if \( start_J < end_J \)
3. then \( A[i] \leftarrow A'[start_J] \)
4. else \( A[i] \leftarrow A'[start_r] \)

The **Split** procedure specified in Alg. 1 implements the decomposition process described in the steps (1) to (4) and (b1) to (b3) of the formal definition. The **Merge** procedure specified in Alg. 2 triggers repeated parallel applications of the **Split** procedure, where every iteration of the while-loop (lines 3 to 7) corresponds to the completion of some single level. Finally the **Map** generates the sorted output by analyzing the sequence of 4-tuples produced by the
Algorithm 2 Pseudocode of the MERGE procedure (CPU-code)

MERGE(A, first1, first2, last)
> u is in A[first1 : first2 - 1], v is in A[first2 : last - 1]
> Idx is an aux. array of 4-tuples with size(Idx) = size(A)
> A’ is an auxiliary array with size(A’) = size(A)

1. Idx[0] ← (first1, first2, first2, last)
2. \( x \leftarrow 1 \)
3. \( \text{while } x < \text{last} - \text{first1} \)
4. \( \text{do seg_size } \leftarrow (\text{last} - \text{first1})/x \)
5. \( \text{for all } i \in \{0, \ldots, x - 1\} \)
6. \( \text{do } \text{Split}(A, \text{Idx}, i, \text{seg_size}) \)
7. \( x \leftarrow x \times 2 \)
8. \( A’ \leftarrow A \) \( \triangleright A’ \) becomes a one-to-one copy of \( A \)
9. \( \text{for all } i \in \{0, \ldots, \text{last} - \text{first1} - 1\} \)
10. \( \text{do } \text{Map}(A, A’, \text{Idx}, i) \)

repeated calls of Split.

The Split and Map procedures are implemented as “kernels” that are executed by the GPU. The Merge procedure delivers the framework for merging and is executed by the host CPU. The for-all-statements in line 5 and 10 become kernel calls where the variable \( i \) becomes some form of global thread identification. So, every single split-operation as well as each single element-map-operation correspond to the execution of a single thread.

4. Integration of monolithic segment mergers - Switching to a two-stage approach

As already stated by Corollary 2 we get for an input of size \( 2^l (l \geq 0) \) on level \( i (l > i \geq 0) \) \( 2^i \) segments of size \( 2^{l-i} \). In the context of our technical realization this means that we have after \( i \) iterations a sequence of 4-tuples, where every 4-tuple specifies the boundary of two input segments that comprise \( 2^{l-i} \) elements together. By multiplying the sequence number \( j \) (\( 2^l > j \geq 0 \)) of some 4-tuple \( t \) by \( 2^{l-i} \) we can compute the start point of the merged outcome for \( t \) in the final output. This property can be exploited as follows:

Without loss of generality we assume that our input has size \( 2^l \) for \( l \geq 0 \). We stop splitting process on level \( i (l > i \geq 0) \), so that each 4-tuple keeps the input locations for an output segment of fixed size \( s = 2^{l-i} \). Then we call \( 2^i \) times some form of segment merger. Each segment merger reads elements from two sections of the input (the boundaries are determined by its 4-tuple) and writes the merged outcome starting at a position that the segment merger computes on the foundation of its sequence-number (which is technically its global thread-id). Figure 5 describes this process graphically for the case that the splitting comprises 3 levels. After the splitting we get a sequence of \( 8 \) 4-tuples that are used by \( 8 \) segment mergers, denoted by \( M_j (2^3 > j \geq 0) \), for creating the sorted output. Because the merging happens now by two separated steps we denote this kind of merging two-stage approach.

Regarding the algorithmic structure of segment mergers exists a variety of different approaches. Three of these are:

(i) The “standard algorithm” as described e.g. in [11]: Starting at the leftmost positions of either input locations we compare pairs of elements, called head elements, such that the smaller element is written to the output and its successor replaces it as head element. This technique preserves stability, but does not comprise any inherent form of parallelization.

(ii) Element Ranking: We compute for all elements the rank in the respective opposite input sequence using binary searches. Knowing this rank we can infer an element’s position in the sorted output sequence. This technique is thoroughly investigated in [7]. It preserves stability and promotes parallelization, but it is non-linear regarding the number of necessary comparisons.

(iii) Bitonic mergers as proposed in [12]: Here we have to read one input sequence reversely for getting bitonically formed inputs. In order to preserve the overall asymptotic optimality regarding comparisons we assume that a bitonic merger is replaced by a simple block copy if its left or right input sequence is empty. Bitonic mergers boost parallelization, but they are neither stable nor linear regarding the number of comparisons or assignments.

![Fig. 5](image-url)  
Graphical description of the two-stage approach.
Corollary 10. If the segment size \( s \) stays constant for all input sizes \( m+n \), then the non-linearities included in some of the above segment mergers do not disturb the overall asymptotic optimality regarding the number of comparisons.

All non-linearities disappear in the asymptotic constants.

5. Benchmarking

For benchmarking purposes we implemented all three variants of the two-stage approach described in Sect. 4. The coding and runtime measurements were done by using the Visual C++ 2008 compiler together with the NVIDIA CUDA SDK 4.0, where all kernel code was implemented in the C-dialect of CUDA. Our hardware equipment consisted of a mainboard based on the AMD 780G chipset that was equipped with an AMD Athlon\textsuperscript{TM} Dual Core Processor 4850e operating at 2.5 Ghz. On GPU-side we used an up-to-date graphic card with NVIDIA GeForce GTX 560 TI chipset, 900 Mhz Core Clock and 1 GB RAM. All benchmarking had been done with the O2-optimization switched on, so the CPU related code was optimized for receiving “maximal speed”. If not mentioned otherwise, the runtimes are measured on the foundation of uniformly distributed 32 bit random numbers of type integer and does not include times for the Host\rightarrow\text{GPU} data exchange.

Figure 6 shows the outcome of our experimental work. The graph labeled Sequential corresponds to the first form of block mergers (implementing the standard algorithm) suggested in Sect. 4. The graph labeled Ranking corresponds to the ranking approach (see (2) in Sect. 4) and the graph labeled Bitonic shows runtimes for bitonic block mergers (see (3) in Sect. 4). The blocksizes used in the context of the benchmarking are shown in the table that is part of Fig. 6.

The memory hierarchy of GPUs was always exploited in the form that block-merging kernels first copied the data to be merged into shared memory before doing their core task on the foundation of these local copies. Finally they wrote the merged outcome back to GPU-side main memory. For reference purposes we included runtimes for the thrust::merge-function (labeled thrust::merge in Fig. 6), which is part of the Thrust library [13].

Dia. (a) informs about the behavior for different degrees of asymmetry with the input-sizes. According to Dia. (a) all merging implementations show some impact of the upper-bound proved in Sect. 2.2. Apart from its odd behavior for strongly asymmetric inputs, the thrust::merge-function is leading for inputs in the range of \( 2^{15} \) to \( 2^{20} \) elements (size of the left sequence) before it loses its competitiveness completely with increasing symmetry of the input sizes. For symmetric inputs, it is even surpassed by the 2-stage approach with sequential block mergers. Dia. (b) inspects this aspect more in detail and contains runtimes for symmetrically sized inputs over a broader range of input sizes. It allows the following observations: (1) Starting from \( 2^{12} \) elements the thrust::merge-function is generally almost three times slower than the stable 2-stage approach on the foundation of rank computations. (2) Bitonic block-mergers, these represent an unstable approach, are even at bit faster than rank computing block-mergers. This is not surprising on the background of their well known efficiency. (3) Block mergers that implement the standard algorithm (sequential block mergers) can not compete. Their behavior becomes even worse for block-sizes larger or smaller than 8 elements.

![Benchmarking for merging.](Fig. 6)
6. Practical Evaluation Using MergeSort

In order to get an impression of the practical value of the above merging strategy we constructed an iterative form of MergeSort on its top and did some benchmarking.

In the following we assume once again that all inputs have some size $2^k$ with $k > 0$. Our iterative MergeSort starts on the foundation of a sequence of sorted tiles of size $2^k$ with $k \geq i > 0$. We get these tiles as result of parallel presortings using e.g. Batcher’s Odd-Even-MERGE [1]. By merging adjacent neighbors of these $2^{k-i}$ many tiles of size $2^k$ we get $2^{k-i-1}$ many sorted tiles of size $2^{i+1}$ each. We continue so until we reach the top level with 2 tiles of length $2^0$. In a final step we merge these two for getting the overall sorted output.

Figure 7 shows a schematic description of the above process in the context of our two stage merging approach. All boxes labeled $P$ represent a Presorter, this is a thread that creates a sorted segment (tile) as output. Boxes carrying the label $S$ represent a splitting process as described in Sect. 2. Each box labeled $B$ denotes a BlockMerger, some thread that reads elements from two input locations and delivers a sorted sequence as output (see Sect. 4). So, in accordance with the two stage approach described in Sect. 4 each level of our iterative MergeSort consists of several parallel splittings followed by a parallel block-merging.

A Pseudocode description of our MergeSort is given in Alg. 3. $BLK\_SIZE$ is a constant that represents the size of all blocks occurring in the context of presortings as well as block mergings ($BLK\_SIZE$ has to be some value $2^k$ with $k \geq i > 0$). All four for-all-do-constructs correspond to the call of a GPU-kernel as already described in Sect. 3. $Idx$ is an array of 4-tuples of size $size(A)/BLK\_SIZE$ that keeps start- and end-positions as it is necessary in the context of the splitting process. Its initialization happens in line 6 to line 11 according to the scheme described above. For example: If we start with 4 consecutive blocks, each consisting of $2^{k-2}$ elements (this is the situation on depth 2 of the iterative MergeSort, i.e. we have 2 parallel merges), then we initialize $Idx$, so that $Idx[0] = (0,2^{k-2},2^{k-2},2^{k-1})$ and $Idx[2^{k-1}/BLK\_SIZE] = (2^{k-1},3\cdot2^{k-2},3\cdot2^{k-2},2^k)$. The variable $size_\_S$ keeps the size of the splitters $S$. This size doubles with the completion of each level as reflected by line 22. The central splitting process is achieved by the code in the lines 12 to 18, where the implementation of the Split procedure is given by Alg. 1. The BlockMerge kernel in line 20 performs block-mergings as described in Sect. 4. It reads from array $A$ and writes to array $A'$ (two arrays are necessary because the read sections - determined by the splitter outcome - and the write section - determined by a blocks position - are normally different). The array copy in line 21 is quite inefficient and was chosen due to formal simplicity. It can be easily replaced by switching references, if $A$ and $A'$ are interpreted as references to arrays.

6.1 Benchmarking for the Iterative MergeSort

We compared the above form of MergeSort with the GPU optimized QuickSort algorithm proposed in [14] (labeled $GPU\_QSort$ in Fig. 8) as well as two GPU oriented Merge-
Sorting implementations. As first MergeSort (labeled CUDA MS in Fig. 8) we chose the sorting example that is part of the CUDA 4.0 SDK. This MergeSort is constructed on the foundation of Batcher’s Odd-Even-Merge [1] for presorting, a ranking technique for block construction and bitonic mergers for block merges. Due to the instability of Odd-Even-Merge as well as bitonic mergers this variant is unstable. As second MergeSort (labeled thrust MS in Fig. 8) we selected the stable MergeSort implementation that is part of the Thrust library [13]. A detailed description of this GPU optimized sorting approach can be found in [3].

For our split based MergeSort we investigated two variants:

1. An unstable variant (labeled unstable MS in Fig. 8) on the foundation of sorting networks (Odd-Even-Merge for presorting and bitonic networks for block merges).
2. A stable variant (labeled stable MS in Fig. 8) that relies on stable merge-based odd-even presorter (as described e.g. in [3]) and the stable ranking approach (see Sect. 4) for all block merges.

As block size $BLK\_SIZE$ (see Alg. 3) we generally chose 256 elements. The benchmarking environment was identical to the one mentioned in Sect. 5. In the context of block mergers the memory hierarchy of GPU’s was exploited as already described in Sect. 5. The input sequences consisted of randomly generated sequences of numbers.

Figure 8 shows the outcome of our benchmarking expressed in sorting rates. The implementations of GPU QuickSort as well as the sorting example included in the CUDA SDK were only available for 32-bit integers, so they are excluded from Dia. (b) and (c) in Fig. 8.

The benchmarking shows a general superiority of our stable as well as unstable MergeSort in comparison to the GPU QuickSort algorithm of [14] as well as the MergeSort that is part of the Thrust library. For up to $2^{17}$ elements the unstable MergeSort originating from the CUDA SDK shows the best performance, before it falls even behind our stable split-based variant starting from $2^{21}$ elements (see Dia. (a)).

According to this design the outer while-loop in Alg. 2 is realized as CPU code and the inner for-all-construct as kernel calls. So, the outer while-loop requires repeated CPU-GPU interactions, which are quite expensive in the context of small input sequences, but are negligible in the context of large sequences. (Alternatively, for the price of an increased code complexity, the for-all-construct could be moved to the outer level and the while-loop could become an inner construct. In this case both iterations are preformed on GPU-side.)

According to Dia. (b) and Dia. (c) the superiority of our approach extends to floating point values as well as 64-bit integers. The generally improved performance for 32-bit floating-point values in comparison with 32-bit integers reflects some special property of Nvidia’s Fermi architecture and has nothing to do with the algorithms itself.

The overall performant behavior of our MergeSort can be explained by the balancing that occurs implicitly in the context of the splitting. After the splitting all block-mergers work on the foundation of equally sized inputs. This is in turn helps avoiding idle threads.

7. Conclusion

We introduced a family of GPGPU oriented merging algorithms that are asymptotically optimal regarding the number of required comparisons. A central feature of our algorithms is the utilization of a technique of repeated symmetric splittings for decomposing a given merging problem into many equally sized independent subproblems. Both, the splitting process itself as well as the solution of all subproblems, can be well parallelized, where the equality of all subproblem with respect to their size implicitly contributes an advanta-

![Fig. 8](image-url) Sorting rates expressed in $10^6$ elements (y-axis) for sorting of sequences of size $2^n$ (x-axis).
geous balancing. Using benchmarking we could show, that our algorithmic framework allows the construction of merge functions that deliver up to 4 times better runtimes than the thrust::merge function of the Thrust library [13].

In order to get a better impression of the practical value of our merging approach we constructed and investigated an iterative MergeSort on top of it. The benchmarking for this MergeSort implementation proved an excellent runtime behavior. Our MergeSort always outperformed the stable MergeSort implementation that is part of the Thrust library as well as the GPU optimized QuickSort variant proposed in [14]. Starting form $2^{18}$ elements it outperformed the unstable MergeSort contained in the example collection of the CUDA 4.0 SDK as well.

The proposed algorithm can be easily extended so that the merging can be distributed among several GPUs. For this purpose the splitting has to start on CPU side and continues until it delivers segments of some predetermined size. Afterwards the merging continues on GPU-side where each GPU gets its data on the foundation of the outcome of the splitting. This technique can also be used for merging huge chunk of data that does not fit into the memory available on GPU-side. Here the CPU first splits into equally sized segments and afterwards these segments are processed one by one on GPU-side.

References

[1] D.E. Knuth, The Art of Computer Programming, Addison-Wesley, 1973.
[2] E. Lindholm, J. Nickolls, S.F. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” IEEE Micro, vol.28, no.2, pp.39–55, 2008.
[3] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore GPUs,” IPDPS 2009, pp.1–10, IEEE, 2009.
[4] T. Hagerup and C. Rüb, “Optimal merging and sorting on the erew pram,” Inf. Process. Lett., vol.33, no.4, pp.181–185, 1989.
[5] F. Gavril, “Merging with parallel processors,” Commun. ACM, vol.18, no.10, pp.588–591, 1975.
[6] R. Cole, “Parallel merge sort,” SIAM J. Comput., vol.17, no.4, pp.770–785, 1988.
[7] D.Z. Chen, “Efficient parallel binary search on sorted arrays, with applications,” IEEE Trans. Parallel Distrib. Syst., vol.6, no.4, pp.440–445, 1995.
[8] R.S. Francis and I.D. Mathieson, “A benchmark parallel sort for shared memory multiprocessors,” IEEE Trans. Comput., vol.37, pp.1619–1626, Dec. 1988.
[9] P.S. Kim and A. Kutzner, “Stable minimum storage merging by symmetric comparisons,” Algorithms - ESA 2004, ed. S. Albers and T. Radzik, Lecture Notes in Computer Science, vol.3221, pp.714–723, Springer, 2004.
[10] K. Dudzinski and A. Dydek, “On a stable storage merging algorithm,” Information Processing Letters, vol.12, no.1, pp.5–8, Feb. 1981.
[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, 2nd ed., MIT Press, 2001.
[12] K.E. Batcher, “Sorting networks and their applications,” AFIPS Spring Joint Computing Conference, AFIPS Conference Proceedings, vol.32, pp.307–314, Thomson Book Company, Washington D.C., 1968.
[13] Thrust library, “http://code.google.com/p/thrust/”
[14] D. Cederman and P. Tsigas, “A practical quicksort algorithm for graphics processors,” ESA, ed. D. Halperin and K. Mehlhorn, Lecture Notes in Computer Science, vol.5193, pp.246–258, Springer, 2008.

Arne Kutzner is a professor of the Department of Information Systems at Hanyang University.

Pok-Son Kim is a professor of the Department of Mathematics at Kookmin University.

Won-Kwang Park is a professor of the Department of Mathematics at Kookmin University.