1 Slide 3: Introduction multi-way sorting algorithms

A general divide-and-conquer technique can be described in three steps: the input is recursively split into k tiles while the tile size exceeds a fixed size M, individual tiles are sorted independently and merged into the final sorted sequence. Most divide-and-conquer algorithms are based either on a k-way distribution or a k-way merge procedure. In the former case, the input is split into tiles that are delimited by k ordered splitting elements. The sorted tiles form a sorted sequence, thus making the merge step superfluous. As for a k-way merge procedure, the input is evenly divided into $\log_k n/M$ tiles, that are sorted and k-way merged in the last step. In contrast to two-way quicksort or merge sort, multi-way approaches perform $\log_k n/M$ scans through the data (in expectation for k-way distribution).

This general pattern gives rise to several efficient manycore algorithms varying only in the way they implement individual steps. For instance, in a multicore gcc sort routine [10], each core gets an equal-sized part of the input (thus k is equal to the number of cores), sorts it using introsort [6], and finally, cooperatively k-way merges the intermediate results.

2 Slide 4: NVidia Tesla Architecture

Current NVidia GPUs feature up to 30 streaming multiprocessors (SMs) each of which containing 8 scalar processors (SPs), i.e., up to 240 physical cores. However, they require a minimum of around $5\,000$–$10\,000$ threads to fully utilize hardware and hide memory latency. A single SM has 2048 32-bit registers, for a total of 64 KB of register space and 16 KB on-chip shared memory that has very low latency and high bandwidth similar to L1 cache.

3 Slide 5: Computing Unified Device Architecture Model

The CUDA programming model provides the means for a developer to map a computing problem to such a highly parallel processing architecture. A common design pattern is to decompose the problem into many data-independent sub-problems that can be solved by groups of cooperative parallel threads, referred to in CUDA as *thread blocks*. Such a two-level parallel decomposition maps naturally to the SIMT architecture: a block virtualizes an SM processor and concurrent threads within the block are scheduled for execution on the SPs of one SM.

A single CUDA computation is in fact similar to the SPMD (single-program multiple-data) software model: a scalar sequential program, a *kernel*, is executed by a set of concurrent threads, that constitute a grid of blocks. Overall, a CUDA application is a sequential CPU, *host*, program that launches kernels on a GPU, *device*, and specifies the number of blocks and threads per block for each kernel call.

4 Slide 6: Performance Guidelines

To achieve peak performance, an efficient algorithm should take certain SIMT attributes into careful consideration:

Conditional branching: threads within a warp are executed in an SIMD fashion, i.e., if threads diverge on a conditional statement, both branches are executed one after another. Therefore, an SIMT processor realizes its full efficiency when all warp threads agree on the same execution path. Divergence between different warps, however, introduces no performance penalty;

Shared memory: SIMT multiprocessors have on-chip memory (currently up to 16 KB) for low-latency access to data shared by its cooperating threads. Shared memory is orders of magnitude faster than the global device memory. Therefore, designing an algorithm that exploits fast memory is often essential for higher performance;

Coalesced global memory operations: aligned load/store requests of individual threads of a warp to the same memory block are coalesced into fewer memory accesses than to separate ones. Hence, an algorithm that uses such access patterns is often capable of achieving higher memory throughput.

5 Slide 7: Algorithm Overview

Sample sort is considered to be one of the most efficient comparison-based algorithms for distributed memory architectures. Its sequential version is probably best described in pseudocode. The oversampling factor a trades off the overhead for sorting the splitters and the accuracy of partitioning.

The splitters partition input elements into k buckets delimited by successive splitters. Each bucket can then be sorted recursively and their concatenation forms the sorted output. If M is the size of the input when `SmallSort` is applied, the algorithm requires $\mathcal{O}(\log_k n/M)$ k-way distribution phases in expectation until the whole input is split into n/M buckets. Using quicksort as a small sorter leads to an expected execution time of $\mathcal{O}(n \log n)$.

6 Slide 8 High Level GPU Algorithm Design

In order to efficiently map a computational problem to a GPU architecture we need to decompose it into data-independent subproblems that can be processed in parallel by blocks of concurrent threads. Therefore, we divide the input into $p = \lceil n/(t \cdot \ell) \rceil$ tiles of $t \cdot \ell$ elements and assign one block of t threads to each tile, thus each thread processes ℓ elements sequentially. Even though one thread per element would be a natural choice, such independent serial work allows a better balance of the computational load and memory latency.

A high-level design of a sample-sort's distribution phase, when the bucket size exceeds a fixed size M, can be described in 4 phases corresponding to individual GPU kernel launches,.

Phase 1. Choose splitters.

Phase 2. Each thread block computes the bucket indices for all elements in its tile, counts the number of elements in each bucket and stores this per-block k-entry histogram in global memory.

Phase 3. Perform a prefix sum over the $k \times p$ histogram tables stored in a column-major order to compute global bucket offsets in the output, for instance the Thrust implementation [9].

Phase 4. Each thread block again computes the bucket indices for all elements in its tile, computes their local offsets in the buckets and finally stores elements at their proper output positions using the global offsets computed in the previous step.

At first glance it seems to be inefficient to do the same work in phases 2 and 4. However, we found out that storing the bucket indices in global memory (as in [7]) was not faster than just recomputing them, i.e., the computation is memory bandwidth bounded so that the added overhead of n global memory accesses undoes the savings in computation.

7 Slide 9: Flavor of Implementation Details computing element bucket indices

We take a random sample S of $a \cdot k$ input elements using a simple GPU LCG random number generator that takes its seed from the CPU Mersenne Twister [5]. Then we sort it, and place each a-th element of S in the array of splitters bt such that they form a complete binary search tree with $bt[1] = s_{k/2}$ as the root. The left child of $b[j]$ is placed at the position $2j$ and the right one at $2j + 1$.

To find a bucket index for an element we adopt a technique that originally was used to prevent branch mispredictions impeding instruction-level parallelism on commodity CPUs [7]. In our case, it allows avoiding conditional branching of threads while traversing the search tree. Indeed, a conditional increment in the loop is replaced by a predicated instruction. Therefore, threads concurrently traversing the search tree do not diverge, thus avoiding serialization. Since k is known at compile time, the compiler can unroll the loop, which further improves the performance.

8 Slide 10: Experimental Evaluation

We report experimental results of our sample sort implementation on sequences of floats, 32-bit and 64-bit integers and key-value pairs where both keys and values are 32-bit integers. We compare the performance of our algorithm to a number of existing GPU implementations including: state-of-the-art Thrust and CUDPP radix sorts and Thrust merge sort [8], as well as quicksort [1], hybrid sort [11] and bbsort [2]. Since most of the algorithms do not accept arbitrary key types, we omit them for the inputs they were not implemented for. We have not included approaches based on graphics APIs in our benchmark, bitonic sort in particular [3], since they are not competitive to the CUDA-based implementations listed above.

Our experimental platform is an Intel Q6600 2.4 GHz quad-core machine with 8 GB of memory. We used an NVidia Tesla C1060 that has 30 Multiprocessors, each containing 8 scalar processors, for a total of up to 240 cores on chip. In comparison to commodity NVidia cards, the Tesla C1060 has a larger memory of 4 GB, that allows a better scalability evaluation. We compiled all implementations using CUDA 2.3 and gcc $4.3.2$ on 64-bit Suse Linux 11.1 with optimization level -O3.

We do not include the time for transferring the data from host CPU memory to GPU memory, since sorting is often used as a subroutine for large-scale GPU computations.

For the performance analysis we used a commonly accepted set of distributions motivated and described in [4].

Uniform. A uniformly distributed random input in the range $[0, 2^{32} - 1]$.

Gaussian. A gaussian distributed random input approximated by setting each value to an average of 4 random values.

Bucket Sorted. For $p \in \mathbb{N}$, the input of size n is split into p blocks, such that the first n/p^2 elements in each of them are random numbers in $[0, 2^{31}/p - 1]$, the second n/p^2 elements in $[2^{31}/p, 2^{32}/p - 1]$, and so forth.

Staggered. For $p \in \mathbb{N}$, the input of size n is split into p blocks such that if the block index is $i \leq p/2$ all its n/p elements are set to a random number in $[(2i - 1)2^{31}/p, (2i)(2^{31}/p - 1)]$.

Deterministic Duplicates. For $p \in \mathbb{N}$, the input of size n is split into p blocks, such that the elements of the first $p/2$ blocks are set to $\log n$, the elements of the second $p/4$ processors are set to $\log(n/2)$, and so forth.

9 Slide 11: Experimental Evaluation, Uniform 32-bit integers

Since the majority of GPU sorting implementations are able to sort 32-bit integers we report sample sort's behavior on all distributions listed above. We include hybrid sort results on floats, since it is the only key type accepted by this implementation, and the sorting rates of other algorithms on floats are similar to the ones on integer inputs.

Low length key type allows both implementations of radix sort to outperform all algorithms similar to the 32-bit integer key-value pairs case. While sample sort demonstates the fastest and still robust performance over all other approaches, except for radix sorts. In particular, it is on average more than 2 times faster than quicksort and hybrid sort for uniform distribution. Due to the uniformity assumption, and hence, a reduced computational cost involved, bbsort is competitive, but still outperformed by our implementation. On the other hand side, the performance of bbsort as well as hybrid sort on Bucket and Staggered distributions significantly degrades when compared to the uniform case. Moreover, on the Deterministic Duplicates input, bbsort becomes completely innefficient, while hybrid sort crashes.

Sample sort is robust with respect to all tested distributions and performs almost equally well on all of them. It demonstrates a sorting rate close to constant, i.e., scales almost linearly with the input size. A higher level of parallelism, and hence, a better possibility of hiding memory latency on large inputs dominate the logarithmic factor in the runtime complexity.

10 Slide 12: Experimental Evaluation, Uniform key-value pairs

Since the best comparison-based sorting algorithm, Thrust merge sort, is designed for key-value pairs only, we can fairly compare it to our sample sort only on this input type. On uniformly distributed keys, our sample sort implementation is at least 25% faster, and achieves on average a 68% higher performance than Thrust merge sort. We do not depict all distributions on key-value pairs, but rather mention the worst case behavior of our implementation on sorted sequences. Sample sort is at least as fast as Thrust merge sort, and still is 30% better on average.

Similarly to radix sort on commodity CPUs, CUDPP radix sort is considerably faster than the comparison-based sample and merge sort on 32-bit integer keys. However, on low level entropy inputs, such as Deterministic Duplicates, even for such low length key types, radix sort is outperformed by sample sort.

11 Slide 13: Experimental Evaluation, Uniform 64-bit integers

With the growth of the key length, radix sort's dependence on the binary key representation makes Thrust radix sort (the only implementation accepting 64-bit keys) not competitive to sample sort. On uniformly distributed keys, our sample sort is at least 63% and on average 2 times faster than Thrust radix. On a sorted sequence, which is the input when our implementation performs worst, its sorting rate does not deviate significantly from the uniform case.

32-bit integer keys. Since the majority of GPU sorting implementations are able to sort 32-bit integers we report sample sort's behavior on all distributions listed above. We include hybrid sort results on floats, since it is the only key type accepted by this implementation, and the sorting rates of other algorithms on floats are similar to the ones on integer inputs.

12 Slide 14: Future Trends Fermi architecture

Computational power increases by a factor of 2, but the memory bandwidth doesn’t seem to have significant improvements, therefore multi-way approaches are likely to outperform two-way on the new architectures.

References

[1] D. Cederman and P. Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proc. European Symposium on Algorithms (ESA)*, volume 5193 of *LNCS*, pages 246–258, 2008.

[2] S. Chen, J. Qin, Y. Xie, J. Zhao, and P.-A. Heng. A Fast and Flexible Sorting Algorithm with CUDA. In *ICA3PP*, volume 5574 of *LNCS*, pages 281–290, 2009.

[3] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 325–336, 2006.

[4] D. R. Helman, D. A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm with an Experimental Study. *J. of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[5] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[6] D. R. Musser. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[7] P. Sanders and S. Winkel. Super Scalar Sample Sort. In *Proc. European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.

[8] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proc. Int'l Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[9] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 97–106, 2007.

[10] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-core Standard Template Library. In *Proc. Int'l Conference on Parallel Processing (Euro-Par)*, volume 4641 of *LNCS*, pages 682–694, 2007.

[11] E. Sintorn and U. Assarsson. Fast Parallel GPU-sorting Using a Hybrid Algorithm. *J. of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.