Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization

Willi Geiselmann[1], Adi Shamir[2], Rainer Steinwandt[1,3], and Eran Tromer[2]

[1] IAKS, Arbeitsgruppe Systemsicherheit, Prof. Dr. Th. Beth,
Fakultät für Informatik, Universität Karlsruhe, 76131 Karlsruhe, Germany
{geiselma, steinwan}@ira.uka.de

[2] Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot 76100, Israel
{adi.shamir, eran.tromer}@weizmann.ac.il

[3] On leave to Department of Mathematical Sciences,
Florida Atlantic University, Boca Raton, FL 33431-0991, USA

Abstract. Motivated by the goal of factoring large integers using the Number Field Sieve, several special-purpose hardware designs have been recently proposed for solving large sparse systems of linear equations over finite fields using Wiedemann's algorithm. However, in the context of factoring large (1024-bit) integers, these proposals were marginally practical due to the complexity of a wafer-scale design, or alternatively the difficulty of connecting smaller chips by a huge number of extremely fast interconnects.

in this paper we suggest a new special-purpose hardware device for the (block) Wiedemann algorithm, based on a pipelined systolic architecture reminiscent of the TWIRL device. The new architecture offers simpler chip layout and interconnections, improved efficiency, reduced cost, easy testability and greater flexibility in using the same hardware to solve sparse problems of widely varying sizes and densities. Our analysis indicates that standard fab technologies can be used in practice to carry out the linear algebra step of factoring 1024-bit RSA keys.

As part of our design but also of independent interest, we describe a new error-detection scheme adaptable to any implementation of Wiedemann's algorithm. The new scheme can be used to detect computational errors with probability arbitrarily close to 1 and at negligible cost.

Keywords: factorization, number field sieve, sparse systems of linear equations

1 Introduction

In recent years, various special-purpose hardware implementations of the Number Field Sieve (NFS) algorithm have been proposed for factoring large (e.g., 1024-bit) integers. These devices address two two critical steps of the NFS: the sieving step [1–7] and the linear algebra step [8–11].

This work focuses on the linear-algebra step of the NFS. While the cost of this step seems to have been reduced to below that of the sieving step (for 1024-bit composites) by the most recent proposals [10, 11], practically these designs are not fully satisfactory: they require (various combinations of) extremely large chips, non-local wiring and high-bandwidth chip interconnects, and thus pose significant technological hurdles.

Below we describe a new systolic design for the NFS linear algebra step, and specifically for the matrix-by-vector multiplications which dominate the cost of the Wiedemann algorithm. This design is both more efficient and more realistic than previous ones. In its simplest form, it consists of a one dimensional chain of identical chips with purely local interconnects, which from a practical standpoint makes it an attractive alternative to previous wafer-scale mesh proposals. For higher efficiency it can be generalized to a two-dimensional array of chips, but unlike previous proposals, this device has standard chip sizes, purely local interconnects, and can use standard DRAM chips for some of its components. In addition, the new design is highly scalable: there is no need to commit to particular problem sizes and densities during the chip design phase, and there is no need to limit the problem size to what can be handled by a single wafer. Since a single chip design of small fixed size can handle a wide range of sparse matrix problems (some of which may be related to partial differential equations rather than cryptography), the new architecture can have additional applications, greatly reduced technological uncertainties, and lower initial NRE cost.

Unlike previous routing based proposals, whose complex data flows required simulation of the whole device and were not provably correct, the present device has a simple and deterministic data flow, so that each unit can be simulated independently. This facilitates the simulation and actual construction of meaningful proof-of-concept sub-devices.

We have evaluated the cost of this device for a specific choice of matrix parameters, which is considered a conservative estimate for the matrix size in factoring 1024-bit integers using NFS. The estimated area×time cost is 6.5 lower than the best previous proposal; the concrete cost estimate is 0.4M US$×year (i.e., excluding non-recurring R&D costs, US$ 0.4M buys enough hardware to obtain a throughput of one solved linear algebra instance per year).

The present design adapts efficiently and naturally to operations over any finite field GF(q), since it does not depend on the in-transit pairwise cancellation of values in GF(2). In particular, it can support the new algorithm of Frey [12, 13]. In fact, it can be used with minor modifications over any ground field, such as the rationals or complex numbers.

Section 2 recalls basic facts about Wiedemann's algorithm and its context in the NFS. Section 3 describes the new hardware architecture. In any large-scale computation the handling of faults is crucial; Section 4 presents a particularly efficient error detection scheme, which can also be adapted to other implementations of block Wiedemann. Section 5 gives a preliminary cost analysis for parameters currently considered as plausible for 1024-bit numbers, and compares it to previous proposals.

2 Preliminaries

For an introduction to the NFS algorithm we refer to [14], and for a detailed account to [15]. Here it is sufficient to keep in mind that the overall running time of the NFS algorithm is dominated by the *sieving step* and the *linear algebra step*. In this paper we exclusively consider the linear algebra step, defined as follows. We are given a $D \times D$ matrix A over GF(2), whose columns correspond to *relations* found in the preceding sieving step (after some pre-processing). Our goal is to find a few vectors in the kernel of A, i.e., several sets of relations that sum to the zero vector. This matrix is large but sparse, with a highly non-uniform distribution of row densities. As in previously proposed devices [8–11], we employ the block Wiedemann algorithm [16, 17] for solving sparse systems of linear equations. Basically, the block Wiedemann algorithm reduces the above to the problem of computing sequences of the form

$$Av, A^2v, \ldots, A^tv \tag{1}$$

for some $v \in \mathrm{GF}(2)^D$. Such a sequence can be computed by means of t matrix-by-vector multiplications, where the matrix A remains fixed and the vector varies. Overall, roughly $2D$ such multiplications are needed, divided into $2K$ chains, where $K > 32$ is the *blocking factor*. The resulting products are not explicitly output after each multiplication; depending on the phase of Wiedemann's algorithm, only their inner product with some fixed vectors or their (partial) sums are needed.

Parameters for 1024-bit composites. At present there is considerable uncertainty about the size and density of the matrix one would encounter in the factorization of a 1024-bit composite, for several reasons: freedom in the choice of the NFS parameters, freedom in the application of pre-processing to the matrix (e.g., to cancel out "large primes"), and lack of complete analysis of this aspect of the NFS algorithm. For concreteness and ease of comparison, in the following we shall assume the "large matrix" parameters from [9], namely a size of $D \times D$ for $D \approx 10^{10}$ and density of 100 entries per column. This leaves a generous conservative margin compared to the smaller matrix expected to be produced by TWIRL [4].

For the sake of concreteness, we propose a concrete instance of our architecture where various design parameters are chosen suitable for the above NFS parameters. In the following, these concrete parameters are designated by angular brackets (e.g., $D \langle\!\langle = 10^{10} \rangle\!\rangle$). Section 5 provides additional details and discusses the cost of the device for these parameters.

3 The New Architecture

We shall unravel the architecture in several stages, where each stage generalizes the former and (when appropriately parameterized) improves its efficiency.

3.1 Basic Scheme

The proposed hardware device is preloaded with a compressed representation of the sparse matrix $A \in \mathrm{GF}(2)^{D \times D}$, as will be detailed below. For each multiplication chain, we load the input vector v and iteratively operate the device to compute the vectors $Av, A^2v, \ldots, A^tv$ and output the appropriate sums or inner products.

We begin by describing an inefficient and highly simplified version of the device, to illustrate its high-level data flow.[4] This simplified device consists of $D \langle\!\langle = 10^{10} \rangle\!\rangle$ *stations* connected in a pipeline. The i-th station is in charge of the i-th matrix row, and contains a compressed representation of the $\langle\!\langle \approx 100 \rangle\!\rangle$ non-zero entries in that row. It is also in charge of the i-th entry of the output vector, and contains a corresponding accumulator $W'[i]$.

In each multiplication, the input vector $v \in \mathrm{GF}(2)^D$ is fed into the top of the pipeline, and moves down as in a shift register. As the entries of v pass by, the i-th station looks at all vector entries v_j passing through it, identifies the ones corresponding to the non-zero matrix entries $A_{i,j}$ in row i, and for those entries adds $A_{i,j} \cdot v_j$ to its accumulator $W'[i]$. Once the input vector has passed all stations in the pipeline, the accumulators $W'[\cdot]$ contain the entries of the product vector Av. These can now be off-loaded and fed back to the top of the pipeline in order to compute the next multiplication.

The one-dimensional chain of stations can be split across several chips: each chip contains one or more complete stations, and the connections between stations may span chip boundary. Note that since communication is unidirectional, inter-chip I/O latency is not a concern (though we do need sufficient bandwidth; the amount of bandwidth needed will increase in the variants given below, and is taken into account in the cost analysis of Section 5).

3.2 Compressed Row Handling

Since the matrix A is extremely sparse, it is wasteful to dedicate a complete station for handling each row of A, as it will be idle most of the time. Thus, we partition A into $u \langle\!\langle = 9600 \rangle\!\rangle$ horizontal stripes and assign each such stripe to a single station (see Figure 1). The number of rows per station is $\mu \approx D/u \langle\!\langle = 2^{20} \rangle\!\rangle$, and each station contains μ accumulators $W'[i]$ with i ranging over the set of row indices handled by the station.

Each station stores all the non-zero matrix entries in its stripe, and contains an accumulator for each row in the stripe. As before, the input vector v passes through all stations, but now there are just u of these (rather than D). Since the entries of v arrive one by one, each station implicitly handles a $\mu \times D$ submatrix of A at each clock cycle.

[4] This basic version is analogous to the electronic pipeline-of-adders version of TWINKLE [2], and many of the improvements described in the following have corresponding analogues in the TWIRL architecture [4].

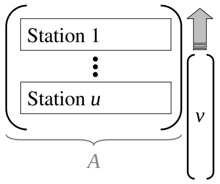

Fig. 1. Distributing the entries of A onto stations

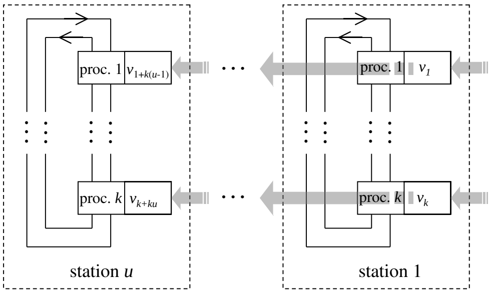

Fig. 2. Subdivision of a chip into stations and processors

3.3 Compressed Vector Transmission

For additional efficiency, we add parallelism to the vector transmission. Instead of each station processing a single entry of v in each clock-cycle, we process v in chunks of $k \,\langle\!\langle = 32 \rangle\!\rangle$ consecutive entries.[5] The inter-station pipeline is thickened by a factor of k. The vector v now passes in chunks of k entries over an inter-station pipeline (in Figure 2 from right to left); in each clock cycle, each station obtains such a chunk from the previous station (to its right), processes it and passes it to the next station (to its left). The first (rightmost) station gets a new part of the vector received from the outside. At each clock cycle, each station now implicitly handles a $\mu \times k$ submatrix of A.

Each station is comprised of k processors, each connected to a separate pipeline line (see Figure 2), and these k processors inside each station are connected via $\gamma \,\langle\!\langle = 2 \rangle\!\rangle$ *intra-station channels*, which are circular shift registers spanning the station. The μ accumulators $W'[i]$ contained in this station are partitioned equally between the k processors.

For processing a k-element chunk of the vector, each of the k processors has to decide whether the vector element v_i it currently holds is relevant for the station it belongs to, i.e., whether any of the μ matrix rows handled by this station contains a non-zero entry in column i. If so, then v_i should be communicated to the processor handling the corresponding accumulator(s) and handled there. This is discussed in the following subsection.

3.4 Processing Vector Elements

Fetching vector elements. The relevance of a vector entry v_i to a given station depends only on i, which is uniquely determined by the clock cycle and the processor (out of the k) it reached. Consequently, each processor needs to read the content of one pipeline line (to which it is attached) at a predetermined set of

[5] The choice of k depends mainly on the number of available I/O pins for inter-chip communication.

clock cycles, specific to that processor, which is constant across multiplications and easily precomputed. This set of cycles is encoded in the processor as follows.

Each processor contains a *fetches table* which instructs it when to read the next vector element from the pipeline. It contains *fetch events*, represented as triplets (τ, f, ℓ) where τ is an δ_{u} $\langle\!\langle= 7\rangle\!\rangle$-bit integer, f is a one-bit flag and ℓ is a $\lceil\log_2(\gamma)\rceil$-bit integer. Such a triplet means: "ignore the incoming vector entries for τ clock cycles; then, if $f = 1$, read the input vector element and transmit it on the ℓ-th intra-station channel".[6] The table is read sequentially, and is stored in compact DRAM-type memory.

Updating the accumulators. Once a relevant vector element v_i has been fetched by some processor and copied to an intra-station channel, we still need to handle it by adding $A_{j,i} \cdot v_i$ to the accumulator $W'[j]$, for every row j handled by this station for which $A_{j,i} \neq 0$. These accumulators (usually just one) may reside in any processor in this station. Thus, each processor also needs to occasionally fetch values from the intra-station channels and process it. Similarly to above, the timing of this operation is predetermined, identical across multiplications and easily precomputed.

To this end, each processor also holds an *updates table* containing *update events* represented as a 5-tuple (τ, f, ℓ, j', x) where τ is an δ_{f} $\langle\!\langle= 7\rangle\!\rangle$-bit integer, f is a one-bit flag, ℓ is a $\lceil\log_2(i)\rceil$-bit integer, j' is a $\lceil\log_2(\mu/k)\rceil$-bit integer and x is a field element.[7] Such a 5-tuple means: "ignore the intra-station channels for τ clock cycles; then, if $f = 1$, read the element $y \in \mathrm{GF}(q)$ currently on channel ℓ, multiply it by x, and add the product to the j'-th accumulator in this processor." This table is also read sequentially and stored in compact DRAM-type memory.

During a multiplication, each processor essentially just keeps pointers into those two tables (which can actually be interleaved in a single DRAM bank), and sequentially executes the events described therein.

An update operation requires a multiplication over $\mathrm{GF}(q)$ and addition of the product to an accumulator stored in DRAM (which is very compact but has high latency). These operations occur at non-regular intervals, as prescribed by the updates table; the processors use small queues to handle congestion, where a processor gets several update events within a short interval. Crucially, the load on these queues is known in advance as a side effect of computing the tables. If some processor is over-utilized or under-utilized, we can change the assignments of rows to stations, or permute the matrix columns, to even the load.

Handling dense rows. All the entries arriving from the intra-station channels while the updated vector is stored into the DRAM have to be held in the processor's queues. As the random-access latency of DRAM is quite large ($\approx$ 70ns), the entries must not arrive too fast. Some of the rows of A are too dense, and could cause congestions of the queues and intra-station channels. To overcome this problem we split such dense rows into several sparser rows, whose sum equals the original. In this way we also ensure that all stations have a similar load and

[6] The flag f is used to handle the cases where the interval between subsequent fetches is more than $2^{\delta_{\mathrm{u}}} - 1$.

[7] Over $\mathrm{GF}(2)$, $x = 1$ always and can thus be omitted.

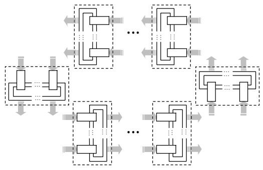

Fig. 3. Arranging the stations into a circle

handle the same number of rows. This increases the matrix size by an insignificant amount ($\langle\!\langle \approx 10^6 \rangle\!\rangle$ additional rows[8]), and the post-processing required to re-combine the split rows is trivial.

Precomputation and simulation. The content of the two tables used by each processor fully encodes the matrix entries. These tables are precomputed once for each matrix A, e.g., using ordinary PCs. Once computed, they allow us to easily simulate the operation of any processor at any clock cycle, as it is completely independent of the rest of the device and of the values of the input vectors. We can also accurately (though inefficiently) simulate the whole device. Unlike the mesh-based approaches in [9–11], we do not have to rely on heuristic run time assumptions for the time needed to complete a single matrix-vector multiplication.

3.5 Skewed Assignment for Iterated Multiplication

In the above scheme, once we have started feeding the initial vector v into the pipeline, after $(D/k) + u$ clock cycles[9] the vector v has passed through the complete pipeline and the vector $A \cdot v$ is stored in the stations. More precisely, each of the u stations contains $\mu = D/u$ consecutive components of v, and we next want to compute the matrix-by-vector product $A \cdot Av$. Thus, we need to somehow feed the computed result Av back into the inter-station pipeline.

To feed the vector Av back into the inter-station pipeline, first we physically close the station interconnects into a circle as depicted in Figure 3; this can be done by appropriate wiring of the chips on the PCB. We also place a memory bank of D/u GF(q) elements at each of the u stations. Collectively, denote these banks by W. At the beginning of each multiplication chain, the initial vector v is loaded into W sequentially, station by station.

During a multiplication, the content of W is rotated, by having each station treat its portion of W as a FIFO of k-tuples: in each clock cycle it sends the last k-tuple of its portion of W to the next station, and accepts a new k-tuple from the previous station. Meanwhile, the processors inside each station function exactly as before, by tapping the flow of k-tuples of vector elements in W at some fixed

[8] Extrapolated from a pre-processed RSA-155 NFS matrix from [18], provided to us by Herman te Riele.

[9] Actually slightly more, due to the need to empty the station channels and processor queues.

point (e.g., the head of the FIFO in that station). Thus, after D/k clock cycles, we have completed a full rotation of the content of W and the multiplication result is ready in the accumulators W. A key point here is that each station sees the contents of W in cyclic order starting at a different offset, but owing to the commutativity of addition in $\mathrm{GF}(q)$ this does not affect the final result.

Having obtained the matrix-by-vector, we can now continue to the next multiplication simply by switching the roles (or equivalently, the contents) of the memory banks W and accumulators W': this amounts to a simple local operation in each processor (note that size and distribution among processors of the cells $W[\cdot]$ and the cells $W'[\cdot]$ is indeed identical). Thus, the matrix-by-vector multiplications can be completed at a rate of one per D/k cycles.

3.6 Amortizing Matrix Storage Cost

Recall that in the block Wiedemann algorithm, we actually execute $2K$ multiplication chains with different initial vectors but identical matrix A. These are separated into two phases, and in each phase we can handle these K chains in parallel. An important observation is that we can handle these K chains using a single copy of the matrix (whose representation, in the form of the two event tables, has so far dominated the cost). This greatly reduces the amortized circuit cost per multiplication chain, and thus the overall cost per unit of throughput.

The above is achieved simply by replacing every field element in W and W' by a K-tuple of field elements, and replacing all field additions and multiplications with element-wise operations on the corresponding K-tuples. The event tables and the logic remain the same. Note that the input and output of each station (i.e., the pipeline width) is now $k \cdot K$ field elements.

3.7 Two-Dimensional Chip Array

As described above, each of the processors inside the station incorporates two types of memory storage: a fixed storage for the representation of the matrix elements (i.e., the event tables), and vector-specific storage (W and W') which increases with the parallelization factor K. Ideally, we would like to use a large K in order to reduce the amortized cost of matrix storage. However, this is constrained by the chip area available for W and W'.

To obtain further parallelization without increasing the chip sizes, we could simply run several copies of the device in parallel. By itself, this does not improve the cost per unit of throughput. But now all of these devices use identical storage for the matrix representation, and access it sequentially at the same rate, so in fact we can "feed" all of them from a single matrix representation. In this variant, the event tables are stored in an external DRAM bank, and are connected to the chips hosting the processors and chain-specific storage through a unidirectional pipeline, as illustrated in Figure 4. Note that communication remains purely local—there are no long broadcast wires.

This variant splits each of the monolithic chips used by the previous variants into a standard DRAM memory chip for matrix storage, plus a chain of small

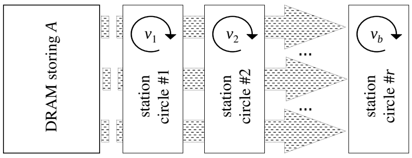

Fig. 4. Using external memory to store the matrix A and b parallel devices, each hosting a circle of stations

ASIC chips for the processors and the storage of the vectors. By connecting $b\,\langle\!\langle= 90\rangle\!\rangle$ such ASIC chips to each DRAM chip, we can increase the blocking factor K by a factor of b without incurring the cost of duplicate matrix storage.

4 Fault Detection and Correction

4.1 A Generic Scheme

To successfully complete the Wiedemann algorithm, the device must compute all the matrix-by-vector multiplications without a single error.[10] For the problem parameters of interest the multiplications will be realized by tens of thousands of chips operating over several months, and it would be unrealistic to hope (or alternatively, expensive to ensure) that all the computations will be faultless. The same concern arises for other special-purpose hardware designs, and also for software implementations on commodity hardware. It is thus crucial to devise algorithmic means for detecting and correcting faults.

A simple real time error-detection scheme would be to apply a linear test: during a preprocessing stage, choose a random $d \times D$ matrix B for an appropriate d, precompute on a reliable host computer and store in the hardware the $d \times D$ matrix $C = BA$, and verify that $Bw' = Cw$ whenever the hardware computes a new product $w' = Aw$. Over GF(q) each row of the matrix B reduces the probability of an undetected error by a factor of q, and thus for $q = 2$ we need at least a hundred rows in B to make this probability negligible. Since each one of the dense $100 \times D$ matrices B and C contains about the same number of 1's as the sparse $D \times D$ matrix A (with one hundred 1's per row), this linear test can triple the storage and processing requirements of the hardware, and meshes poorly with the overall design whose efficiency relies heavily on the sparseness of the matrix rows. Note that we cannot solve this problem by making the $100 \times D$ matrix B sparse, since this would greatly reduce the probability of detecting single bit errors.

In the following we describe an alternative error-detection scheme, which provides (effectively) an arbitrarily small error probability at a negligible cost,

[10] Note the contrast with the NFS sieving step, which can tolerate both false positive and false negative errors in its smoothness tests.

under reasonable assumptions. It inspects only the computed (possibly erroneous) matrix-by-vector products, and can thus be applied to any implementation of Wiedemann's algorithm. We will consider its operation over any finite field $\mathrm{GF}(q)$, though for integer factoring via NFS only $q = 2$ is of interest.

Detection. Let $w_0, w_1, w_2, \ldots \in \mathrm{GF}(q)^D$ denote the sequence of vectors computed by the device, where $w_0 = v$. To verify that indeed $w_i = A^i v$ for all $i > 0$, we employ the following randomized linear test. For a small integer $d \langle\!\langle = 200 \rangle\!\rangle$, choose a single vector $b \in \mathrm{GF}(q)^D$ uniformly at random, and precompute on a reliable computer the single vector $c^{\mathrm{T}} = b^{\mathrm{T}} A^d$ (here $^{\mathrm{T}}$ denotes transpose). After each w_i is computed, compute also the inner products $b^{\mathrm{T}} w_i$ and $c^{\mathrm{T}} w_i$ (which are just field elements). Save the last d results of the latter in a small shift register, and after each multiplication test the following condition:

$$b^{\mathrm{T}} w_i = c^{\mathrm{T}} w_{i-d} \ . \tag{2}$$

If equality does not hold, declare that at least one of the last d multiplications was faulty.

Correctness. If no faults have occurred then (2) holds since both sides equal $b^{\mathrm{T}} A^i v$. Conversely, we will argue that the first faulty multiplication $w_j \neq A w_{j-1}$ will be detected within d steps with overwhelming probability, under reasonable assumptions.

Let us first demonstrate this claim in the simplest case of some transient error ε which occurs in step j. This changes the correct vector $A^j v$ into the incorrect vector $w_j = A^j v + \varepsilon$. All the previous w_i for $i < j$ are assumed to be correct, and all the later w_i for $i > j$ are assumed to be computed correctly, but starting with the incorrect w_j in step j. It is easy to verify that the difference between the correct and incorrect values of the computed vectors w_i for $i > j$ evolves as $A^{i-j}\varepsilon$, and due to the randomness of the matrix A generated by the sieving step these error vectors are likely to point in random directions in the D-dimensional space $\mathrm{GF}(q)^D$. The device has d chances to catch the error by considering pairs of computed vectors which are d apart, with the first vector being correct and the second vector being incorrect. The probability that all these d random error vectors will be orthogonal to the single random test vector b is expected to be about q^{-d}, which is negligible; the computational cost was just two vector inner products per matrix-by-vector multiplication.

The analysis becomes a bit more involved when we assume that the hardware starts to malfunction at step j, and adds (related or independent) fault patterns to the computed result after the computation of each matrix-vector product from step j onwards. Let the result of the i-th multiplication be $w_i = A w_{i-1} + \varepsilon_i$, where the vector ε_i is the error in the output of this multiplication. We consider the first fault, so $\varepsilon_i = 0$ for all $i < j$. Assume that $j \geq d$ ($j < d$ will be addressed below). By the linearity of the multiplication and the minimality of j, we can expand the above recurrence to obtain $w_i = A^i v + \sum_{i'=j}^{i} (A^{i-i'} \varepsilon_{i'})$ $(i \geq j)$. Plugging this into (2) and canceling out the common term $b^{\mathrm{T}} A^i v$, we get that for $j \leq i < j + d$, (2) is equivalent to:

$$b^{\mathsf{T}} r_i = 0 \qquad \text{where} \qquad r_i = \sum\nolimits_{i'=j}^{i} (A^{i-i'} \varepsilon_{i'}) \ . \tag{3}$$

We assume that each error ε_i is one of at most $(qD)^\alpha$ possibilities for some $\alpha \ll D/d$ (e.g., $\langle\!\langle \alpha = 10^5 \rangle\!\rangle$), regardless of A and b. This suffices to enumerate all reasonably likely combinations of local faults (corrupted matrix entries, faulty pipeline connections, errors in $\mathrm{GF}(q)$ multipliers, memory bit flips, etc.). We also make the simplifying (though not formally correct) assumption that $A^{10}, ..., A^{d-1}$ are random matrices drawn uniformly and independently.[11] Then for any fixed values of ε_i, the vectors in the set $R = \{r_i\}_{i=j+10}^{j+d-1}$ are drawn uniformly and independently from $\mathrm{GF}(q)^D$ (recall that $\varepsilon_j \neq 0$), and thus the probability that the span of R has dimension less than $|R| = d - 10$ is smaller than $dq^{-(D-d)}$ (which is a trivial upper bound on the probability that one of the $d - 10$ vectors falls into the span of the others). By assumption, there are at most $(qD)^{\alpha d}$ possible choices of $(\varepsilon_i)_{i=j+1}^{j+d}$. Hence, by the union bound, the probability that the span of R has dimension less than $d - 10$ is at most $(qD)^{\alpha d} \cdot dq^{-(D-d)} = d \cdot q^{\alpha d \log_q D + d - D}$, which is negligible. Conditioned on the span of R having full rank $d - 10$, the probability of the random vector b being orthogonal to the span of R is $q^{-(d-10)}$, which is also negligible. Hence, with overwhelming probability, at least one of the tests (3) for $j + 10 < i < j + d$ will catch the fault in w_j.

Startup and finalization. Note that the test (2) applies only to $i > d$, and moreover that our analysis assumes that the first d multiplications are correct. Thus, for each of the $2K$ multiplication chains of block Wiedemann, we start the computation by computing the first d multiplications on a reliable general-purpose computer, and then load the state (including the queue of $c^{\mathsf{T}} w_i$ values for $i = 0, \ldots, d$) into the device for further multiplications.

Also note that in the analysis, the results of the j-th multiplications are implicitly checked by (2) for $i = j, \ldots, j + d - 1$. Thus, in order to properly check the last d multiplications in each chain, we run the device for d extra steps and discard the resulting vectors but still test (2).

Recovery. The above method will detect a fault within d clock cycles (with overwhelming probability), but will not correct it. Once the fault is detected, we must backtrack to a known-good state without undoing too much work. Assuming a sufficiently low probability of error, it is simplest to dump a full copy of the current vector w_i from the device into a general-purpose computer,

[11] The sieving and preprocessing steps of NFS yield a matrix A that has nearly full rank and is"random-looking" except for some biases in the distribution of its values: A is sparse (with density $\langle\!\langle \approx 100/10^{10} \rangle\!\rangle$) and its density is decreasing with the row number. The first few self-multiplications increase the density exponentially and smoothen the distribution of values, so that A^{10} has full and uniform density. The independence approximation is applicable since we are looking at simple local properties (corresponding to sparse error vectors), which are "mixed" well by the matrix multiplication. While the resulting matrices do have some hidden structure, realistic fault patterns are oblivious to that structure.

at regular but generously-spaced intervals; this can be done by another special station tapping the pipeline. The backup vectors may be stored on magnetic media, and thus their storage has negligible cost. When a fault is detected, the faulty component can be replaced (or a spare device substituted) and the computation restarted from the last known-good backup.

4.2 Device-Specific Considerations

Implementation. The above scheme requires only the computation of two inner products ($b^{\mathrm{T}} w_i$ and $c^{\mathrm{T}} w_i$) for each multiplication. In the proposed hardware device, this is achieved by one additional station along the pipeline, which taps the vector entries flowing along the pipeline and verifies their correctness by the above scheme. This station contains the entries of b and c in sequential-access DRAM. For each of the K vectors being handled, it processes a k-tuple of vector entries at every clock cycle, keeps the d most recent values of $c^{\mathrm{T}} w_i$ in a local FIFO queue at this station, and performs the test according to (2).

Halving the cost. The storage cost can be halved by choosing b pseudorandomly instead of purely randomly; the number of multipliers can also be nearly halved by choosing b to be very sparse.

Using faulty chips. In addition to the above high-level error-recovery scheme, it is also useful to work around local faults in the component chips: this increases chip yield and prevents the need to disassemble multi-chip devices if a fault was discovered after assembly. To this end, the proposed device offers a significant level of fault tolerance due to its uniform pipelined design: we can add a “bypass” switch to each station, which effectively removes it from the pipeline (apart for some latency). Once we have mapped the faults, we can work around any fault in the internals of some station (this includes the majority circuit area) by activating the bypass for that station and assigning its role to one of a few spare stations added in advance. The chip containing the fault then remains usable, and only slightly less efficient.

5 Cost and Performance

5.1 Cost for 1024-bit NFS Matrix Step

As explained in Section 2, there is considerable uncertainty about the size and density of the matrices that would appear in the factorization of 1024-bit composites using the Number Field Sieve. For concreteness and ease of comparison, throughout Section 3 and in the following we assume the rather conservative “large matrix” parameters (see Section 2).

Clearly there are many possibilities for fixing the different parameters of our device, depending on such parameters as desired chip size and number of chips. One may even consider combining the above design with the splitting of the processed matrix into submatrices as put forward in [10], thereby giving up the homogeneity and purely local communication but decreasing the dimension of the vectors that have to be handled. In the following we consider a specific parameter set, which focuses on practicality with today's technology.

We assume 90nm chip manufacturing technology with DRAM-type process[12], a net chip area of 1 cm^2, a per-chip I/O bandwidth of 1024 Gbit/s, and a clock rate of 1GHz. A DRAM access is assumed to take 70 clock cycles. These parameters are quite realistic with current technology.

We employ a 300×90 array of ASIC chips. Each column of 300 chips contains $u = 9600$ stations (32 per chip). Each station consists of $k = 32$ processors, communicating over $\gamma = 2$ intra-station channels, with a parallelization factor of 10. Each of the 300 rows, of 90 chips each, is fed by a 108Gbit DRAM module. Overall, the blocking factor is $K = 10 \cdot 90 = 900$. This array can complete all multiplication chains in ≈ 2.4 months.

The total chip area, including the matrix storage, is less than 90 full 30cm wafers. Assuming a "silicon cost" of US$ 5000 per wafer, and a factor 4 increase for overheads such as faulty chips, packaging, testing and assembly, the total cost is under US$ 2M.

Comparison to previous designs. A mesh-based design as considered in [11], adapted to 90nm technology and using 85×85 chips of size 12.25 cm^2 each, will require about 11.7 months to process the above matrix. The higher complexity of this design limits the clocking rate to 200 MHz only. Comparing throughput per silicon area, the new device is 6.5 more efficient; it also has much smaller individual chips and no need for non-local wiring.

Implications for 1024-bit factorization. With the above device and matrix size, the cost of the NFS linear algebra step is 0.4M US$×year, which is significantly lower than that of the NFS sieving step using the TWIRL device [4]. Moreover, TWIRL is expected to produce a matrix significantly smaller than the conservative estimate used above, so the cost of the linear algebra step would be lower than the above estimate. Since TWIRL, being a wafer-scale design, is also more technologically challenging, this reaffirms the conclusion that at present the bottleneck of factoring large integers is the NFS sieving step [9].

5.2 Further Details

To derive concrete cost and performance estimates for the 1024-bit case, several implementation choices for parameters, such as δ_{u}, δ_{f}, γ, τ, have been determined experimentally as follows. For the above problem and technology parameters, and a large randomly drawn matrix, we used a software simulation of a station to check for congestions in bus and memory accesses, and chose design parameters for which such congestions never occur experimentally. Recall that the device's operation is deterministic and repetitive (see Section 3.4), so the simulation accurately reflects the device's operation with the given parameters.

In the following we briefly mention some aspects of the circuit area and its analysis, as used to derive the above estimate. Note that we employ the

[12] Amortized DRAM density is assumed to be $0.1\mu m^2$ per bit, and the logic is assumed to have an average density of $1.4\mu m^2$ per transistor.

split design of Section 3.7, which puts the matrix storage in plain DRAM chips and the logic and vector storage in ASIC chips. For these parameters, memory storage dominates area: approximately 97% of the ASIC chip area is occupied by the DRAM which stores the intermediate vectors (i.e., W and W'). Thus, the suitable chip production process is a DRAM process optimized for maximum memory density (at the expense of slightly larger logic circuits); similar cases arose in previous proposals [9–11]. Each of the $k \langle\!\langle = 32 \rangle\!\rangle$ processors in each of the 32 stations in each of the 300×90 ASIC chips contains the following logic:

- A K/b-bit register for storing the K/b-tuples of GF(2) elements flowing along from the inter-station pipeline ($\approx 8 \cdot K/b$ transistors).
- A K/b-bit register for each of the $\gamma \langle\!\langle = 2 \rangle\!\rangle$ intra-station channels ($\approx 8 \cdot \gamma \cdot K/b$ transistors).
- A FIFO queue of depth $\langle\!\langle 2 \rangle\!\rangle$ for storing elements arriving on the inter-station pipeline along with the number of the internal bus onto which the respective element is to be written. For this $\approx 2 \cdot 8 \cdot (K + \lceil \log_2(\gamma) \rceil)$ transistors per queue entry are sufficient.
- A FIFO queue of depth $\langle\!\langle 4 \rangle\!\rangle$ for storing elements arriving on the intra-station channels that have to be XORed to the vector. Each entry consists of a K/b-tuple of bits for the vector and a row number in the submatrix handled by the station has to be stored. This occupies $\approx 4 \cdot 8 \cdot (K/b + \lceil \log_2 \lceil D/(ku) \rceil \rceil)$ $\langle\!\langle = 4 \cdot 8 \cdot (10 + 15) \rangle\!\rangle$ transistors per queue entry.

In addition to the registers and queues, we need some logic for counters (to identify the end of a vector and to decide when to read another element from a bus), multiplexers, etc. For the parameters of interest, < 1500 transistors are sufficient for this. Overall, the 32×32 processors on each chip occupy $\langle\!\langle \approx 3.2\text{mm}^2 \rangle\!\rangle$.

The DRAM needed splits into three parts.

- For storing $2 \cdot K/b$ vectors in $\mathrm{GF}(2)^{\lceil D/(uk) \rceil}$,: $2 \cdot K/b \cdot D/(uk)$ bit $\langle\!\langle \approx 650$ Kbit$\rangle\!\rangle$.
- For the fetches table: $\delta_\mathrm{u} + 1 + \lceil \log_2(\gamma) \rceil$ bits per entry.
- For the updates table: $\delta_\mathrm{f} + 1 + \lceil \log_2(\gamma) \rceil + \lceil \log_2 \lceil D/(uk) \rceil \rceil$ bits per entry.

Overall, the DRAM on each chip occupies $\langle\!\langle \approx 67\text{mm}^2 \rangle\!\rangle$. The time for each of the $\approx 2D/K$ matrix-by-vector multiplications is $\approx e + D/k$ clock cycles, where e gives some leeway for emptying queues and internal buses (for the parameters we are interested in $e \ll 1000$ is realistic).

6 Conclusion

We have described a pipelined systolic design for the matrix-by-vector multiplications of the block Wiedemann algorithm, which exhibits several advantages over the prior (mesh-based) approach. It has lower cost and modest technological requirements; specifically, unlike previous proposals it uses standard chip sizes and purely local communication. The architecture is scalable, and offers the flexibility to handle problems of varying sizes. The operation is deterministic and allows local simulation and verification of components. We have also described an efficient error detection and recovery mechanism, which can also be adapted to other software or hardware implementations of Wiedemann's algorithm.

For 1024-bit RSA keys, executing the linear algebra step of the NFS using this device appears quite realistic with present technology, at a cost lower than that of the NFS sieving step.

References

1. Shamir, A.: Factoring Large Numbers with the TWINKLE Device. In: CHES 1999. Volume 1717 of LNCS, Springer (1999) 2–12
2. Lenstra, A.K., Shamir, A.: Analysis and Optimization of the TWINKLE Factoring Device. In: EUROCRYPT 2000. Volume 1807 of LNCS, Springer (2000) 35–52
3. Geiselmann, W., Steinwandt, R.: A Dedicated Sieving Hardware. In: PKC 2003. Volume 2567 of LNCS, Springer (2003) 254–266
4. Shamir, A., Tromer, E.: Factoring Large Numbers with the TWIRL Device. In: CRYPTO 2003. Volume 2729 of LNCS, Springer (2003) 1–26
5. Geiselmann, W., Steinwandt, R.: Yet Another Sieving Device. In: CT-RSA 2004. Volume 2964 of LNCS, Springer (2004) 278–291
6. Franke, J., Kleinjung, T., Paar, C., Pelzl, J., Priplata, C., Stahlke, C.: SHARK - A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers. In: SHARCS 2005. (2005)
7. Franke, J., Kleinjung, T., Paar, C., Pelzl, J., Priplata, C., Simka, M., Stahlke, C.: An Efficient Hardware Architecture for Factoring Integers with the Elliptic Curve Method. In: SHARCS 2005. (2005)
8. Bernstein, D.J.: Circuits for Integer Factorization: a Proposal. At the time of writing available electronically at `http://cr.yp.to/papers/nfscircuit.pdf` (2001)
9. Lenstra, A.K., Shamir, A., Tomlinson, J., Tromer, E.: Analysis of Bernstein's Factorization Circuit. In: ASIACRYPT 2002. Volume 2501 of LNCS, Springer (2002) 1–26
10. Geiselmann, W., Steinwandt, R.: Hardware for Solving Sparse Systems of Linear Equations over GF(2). In: CHES 2003. Volume 2779 of LNCS, Springer (2003) 51–61
11. Geiselmann, W., Köpfer, H., Steinwandt, R., Tromer, E.: Improved Routing-Based Linear Algebra for the Number Field Sieve. In: Proceedings of ITCC '05 – Track on Embedded Cryptographic Systems, IEEE Computer Society (2005) 636–641
12. Frey, G.: A First Step Towards Computations in Brauer Groups and Applications to data Security. Invited talk at WARTACRYPT '04 (2004)
13. Frey, G.: On the Relation between Brauer Groups and Discrete Logarithms. Unpublished manuscript (2004)
14. Pomerance, C.: A Tale of Two Sieves. Notices of the ACM (1996) 1473–1485
15. Lenstra, A.K., Hendrik W. Lenstra, J., eds.: The development of the number field sieve. Volume 1554 of Lecture Notes in Mathematics. Springer (1993)
16. Coppersmith, D.: Solving Homogeneous Linear Equations over GF(2) via Block Wiedemann Algorithm. Mathematics of Computation **62** (1994) 333–350
17. Villard, G.: Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In: International Symposium on Symbolic and Algebraic Computation — ISAAC '97, ACM (1997) 32–39
18. Cavallar, S., Dodson, B., Lenstra, A., Lioen, W., Montgomery, P., Murphy, B., te Riele et al., H.: Factorization of a 512-bit RSA modulus. In: EUROCRYPT 2000. Volume 1807 of LNCS, Springer (2000) 1–17