Implementing a Sieving Algorithm on a Dynamic Reconfigurable Processor (Extended Abstract) ⋆

Takeshi Shimoyama, Tetsuya Izu, and Jun Kogure

FUJITSU Limited,
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan
{shimo-shimo,izu,kogure}@jp.fujitsu.com

Abstract. This extended abstract proposes an efficient implementation of the sieving step in the number field sieve method of integer factorization on a dynamic reconfigurable processor "DAPDNA-2". The pipeline method and the bucket sorting method for the line sieving will be discussed.
Keywords: hardware implementation, dynamic reconfigurable processor, DAPDNA-2, integer factoring, sieving algorithm

1 Introduction

The integer factoring problem is one of the most fundamental topics in the area of cryptology since the hardness of this problem assures the security of some public-key cryptosystems. The number field sieve method (NFS) [7] is known as the most efficient algorithm for factoring large composite integers. In 2003, Franke et al. factored a 576-bit integer (RSA576) by NFS, which is the current world record of the factorization. Since the complexity of NFS grows subexponential with regard to the integer size, it is widely believed that factoring 1024-bit integers is infeasible in next 10 years by the same software approach. It is natural to consider a special hardware dedicated to integer factorizations.

NFS consists of four major steps, the polynomial selection step, the sieving step, the linear algebra step, and the square root step. Among them, the sieving step and the linear algebra step are theoretically and experimentally dominant steps. In 2001, Bernstein employed a special hardware design for the linear algebra step based on a sorting algorithm with standard ASIC architectures [2]. Then Lenstra et al. enhanced the device by using a routing algorithm [9]. Geiselmann and Steinwandt applied these ideas to the sieving step and proposed two designs [4, 5]. On the other hand, Shamir and Tromer improved an optical sieving device TWINKLE [10] into a novel ASIC-based hardware TWIRL [11]. By these contributions, it is expected that the linear algebra step is easily processed compared to the sieving step.

⋆ This work is financially supported by a consignment research from the the National Institute of Information and Communications Technology (NICT), Japan.

However, all these designs are just theoretical; no experimental results have been known up to the present. One of the major reason may be that designing and manufacturing such devices require a quite a few amount of money and time. In this extended abstract, we proposed an efficient implementation of the sieving step on a dynamic reconfigurable processor "DAPDNA-2" [6]. This processor has two features: one is flexibility of the programming (like usual PC) and the other is high-performance of the process (like FPGA). The performance is not as excellent as that of ASIC, however, it is quite suitable for test hardware implementations. Considering such features of DAPDNA-2, we discuss the line sieving of the number field sieve method. Since it is just the beginning of the project, we only deal with these basic algorithms. Discussions and optimizations of more sophisticated algorithms will be future work.

2 Dynamic Reconfigurable Hardware "DAPDNA-2"

A dynamic reconfigurable processor DAPDNA-2 is introduced and manufactured by IPFlex [6]. A chip DAPDNA-2 has dual-core processors DAP and DNA. DAP is a high-performance RISC processor core, and DNA is a two-dimensional array of 376 processing elements (PEs) (see Figure 1). DAPDNA-2 has a circuit configuration within a chip and able to switch the configuration dynamically, allowing multiple applications. This processor enables to deal with processes implemented on multiple chips conventionally on a single chip.

DAPDNA-2 can be configured to provide the optimal circuitry for a particular application. This configuration takes place not only when the system is initialized, but can also occur dynamically in a single clock cycle [operating at 166MHz] while the system is running, to meet the instantaneous needs of the applications implemented by the system. DNA configuration data are stored in four banks, one foreground bank, three background banks. Additional configuration data can be loaded from external memory.

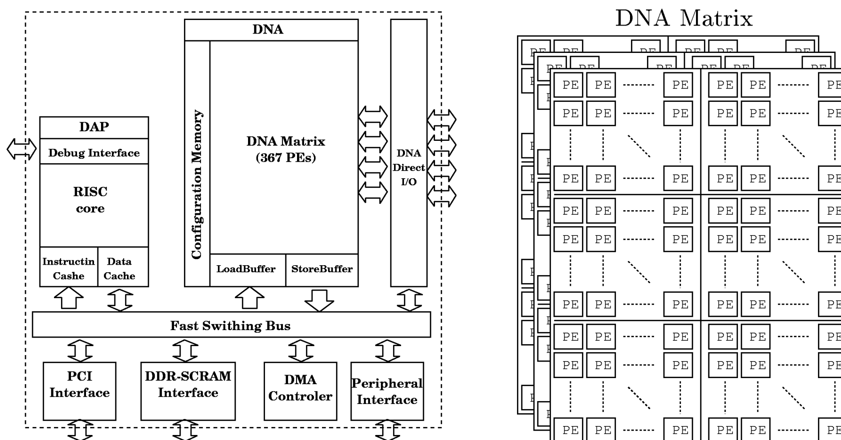
3 Sieving on DAPDNA-2

Let N be a large composite integer we are going to factor. Let $f(x) \in \mathbb{Z}[x]$ be an irreducible polynomial and m be an integer such that $f(m) \equiv 0 \pmod{N}$. We suppose that the polynomial $f(x)$ and the integer m are given by certain means for the input N. Main purpose of the sieving step is to collect a huge number of relations from a given domain $[-H_a, H_a] \times [1, H_b] \subset \mathbb{Z}^2$. Here a pair $(a, b) \in \mathbb{Z}^2$ is called the relation if it satisfies all of the following conditions:

- $\gcd(a, b) = 1$,
- $N_{\mathrm{R}}(a, b) = |a + bm|$ is B_{R}-smooth for a given parameter B_{R},
- $N_{\mathrm{A}}(a, b) = |(-b)^{\deg f} f(-a/b)|$ is B_{A}-smooth for a given parameter B_{A}.

An integer x is called y-smooth if all prime factors of x is less than or equal to y. We also suppose that a set of primes less than or equal to B_{R} (B_{A}) is prepared as the factor base.

Fig. 1. Block Diagram of DAPDNA-2



Algorithm 1 *Line sieving*

1: for $b \leftarrow 1$ *to* H_b
2: *set* $S[a]$ *to* $\log N_{\mathrm{R}}(a, b)$ *for all* a
3: *for prime* $p \leftarrow 2$ *to* B_{R}
4: *compute the sieving point* $a \geq -H_a$
5: *while* $a < H_a$
7: $S[a] \leftarrow S[a] - \log p$
8: $a \leftarrow a + p$

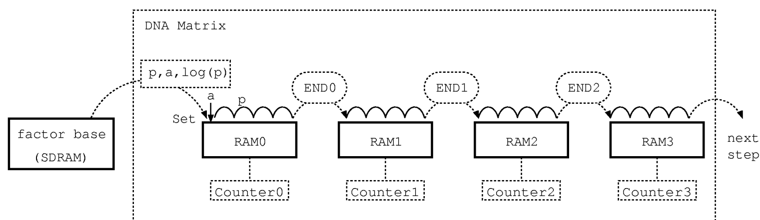
3.1 Pipeline Method

Algorithm 1 is a straight-forward algorithm for the sieving step (line sieving). When we process Algorithm 1 on usual PC by software, step 7 in Algorithm 1 is dominant since simultaneous memory access cannot be processed in general. On the other hand, simultaneous memory access can be processed to different RAM-PE in the DNA matrix in DAPDNA-2 independently. Using this property of DAPDNA-2, we implement and optimize step 7 in Algorithm 1 as in the followings.

Remark. DNA matrix has 32 RAM-PEs. We regard these RAM-PEs as a sequential memory space. Sieving length is chosen to fit this memory space for each sieving.

In order to process the line sieving efficiently, we use the pipeline method for the line sieving. A model of the pipeline method is shown in Figure 2. Suppose the sieving points a and the information on $\log p$ for each prime p are stored in

Fig. 2. Pipeline Algorithm



Algorithm 2

- *On RAM0*
  1. *When END0 flag is "off" and Counter0 is passive,*
     set Counter0 as active,
     read a data $(p, a, logp)$ *from the factor base in SDRAM, and*
     subtract $\log p$.
  2. *When access address is beyond RAM0 and subtractions are finished,*
     set Counter0 as passive and END0 flag as "on".
- *On RAM1,*
  1. *When END0 flag is "on", END1 flag is "off" and Counter1 is passive,*
     set Counter1 as active and END0 flag as "off", and
     subtract $\log p$.
  2. *When access address is beyond RAM1 and subtractions are finished,*
     set Counter1 as passive and END1 flag as "on".
- *On RAM2, process as on RAM1.*
- *On RAM3,*
  1. *When END2 flag is "on" and Counter3 is passive,*
     set Counter3 as active, END2 flag as "off", and
     subtract $\log p$.
  2. *When access address is beyond RAM3 and subtractions are finished,*
     set Counter3 as passive.

SDRAM. In Figure 2, RAM0, ..., RAM3 is an access unit in the pipeline. Each RAMi ($i = 0, \ldots, 3$) is not limited to a single RAM-PE, but also a combination of some RAM-PEs. Counter$i$ ($i = 0, 1, 2, 3$) implies a memory address to be subtracted by $\log p$ for RAMi. Here Counter$i$ is active when the subtraction by $\log p$ is processed, or passive otherwise. ENDi ($i = 0, 1, 2$) is a 1-bit flag which indicates whether the sieving is finished ("on") or not ("off") for RAMi.

Algorithm 2 shows a concrete algorithm for the pipeline method. In order to process the pipeline sieving efficiently, the next factor base should be read just after finishing the subtractions in RAM0. To do so, Counter1 should be passive and END1 flag should be "off" when END0 flag is "on". Conversely, when END0 flag is "on", if Counter1 is active or END1 flag is "on", new factor base is not

Algorithm 3

1: Let all buckets empty
2: **for prime** p *in factor base*
3: *Compute* $a \geq -H_a$ *as the first sieving point*
4: **while** $a < H_a$
5: *Store* $(a, \log p)$ *to the* $\left\lfloor \dfrac{a + H_a}{r} \right\rfloor$*-th bucket*
6: $a \leftarrow a + p$

Algorithm 4

1: **for** *all buckets that are numbered* i $(0 \leq i < n)$
2: **for** *all* $(a, \log p)$ *in the bucket* i
3: $\mathtt{S}[a] \leftarrow \mathtt{S}[a] + \log p$

read and pipeline stall would be occurred. To avoid such pipeline stalls, sieving processes for a factor base in every RAMi should be finished before sieving processes for the next factor base will be finished.

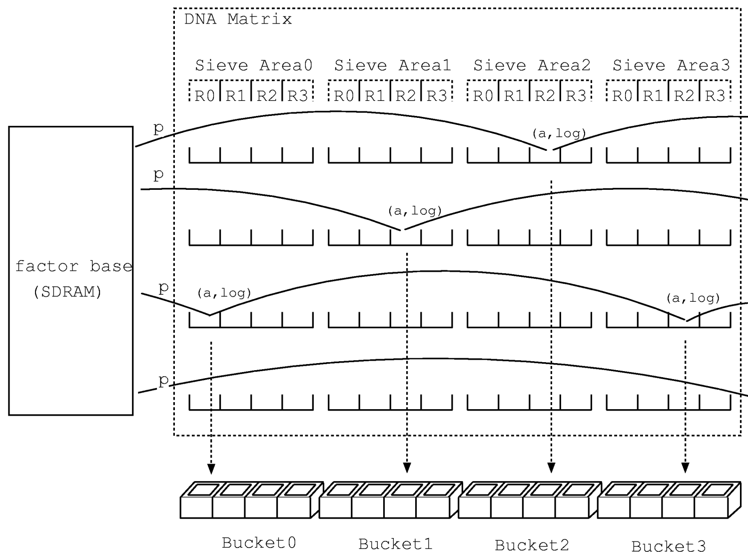
In general, the number of subtractions by $\log p$ for a fixed sieving area is determined by the size of primes in the factor base and thus the number would be large when p is small. So the pipeline is efficient if a factor base is lined in descending order.

In addition, when the size of a prime in a factor base is larger than the size of the sieving area (for example, a sum of RAMi ($i = 0, \ldots, 3$) in Figure 2), subtractions by $\log p$ are not always processed and pipelining become meaningless. Thus it would be efficient to adapt pipelining for primes whose sizes are smaller than the sieving size. In order to deal with larger primes, the bucket sorting method for the sieving proposed by Aoki and Ueda will be applied in the next section.

3.2 Bucket Sorting Method

The pipeline method described in the previous section was efficient for only small primes. In order to process the line sieving better than the pipeline method, namely for larger primes, Aoki and Ueda proposed a new algorithm using bucket sorting [1]. The algorithm that cleverly uses the cache memory on PC, accelerates the memory update processes in the sieving step to several times faster than that of the simple $\log p$ updating. Algorithm 3 throws the pairs $(a, \log(p))$ of a sieving point and an update value in the buckets, and Algorithm 4 updates array $\mathtt{S}[a]$ using the elements in the buckets. The sieve area is $-H_a \leq a \leq H_a$. Let n be the number of buckets, and r be $\left\lceil \dfrac{n_{\mathtt{S}}}{n} \right\rceil$, where $n_{\mathtt{S}}$ denotes the number of elements in $\mathtt{S}$. For the detail of the algorithm, please refer to [1].

Fig. 3. Bucket Sort Sieving



Our sieving process consists of the following three parts;

1. computation of sieve point using bucket sort (bucket sort part),
2. memory update by using sorted sieving points (sieving part),
3. comparing the value in the memory and threshold value. (check part).

Bucket sort part

Figure 3 shows an outline of the implementation of the bucket sorting method on DAPDNA-2. This figure describes the pre-process by using the bucket sort for 4 sievings at a time. Area0,...,Area3 are virtual memory areas with size n_S for each, and Area$i$ is divided into inner virtual memory areas Rj ($j = 0, \ldots, 3$). Bucket0,...,Bucket3 collects information on the subtractions by $\log p$ corresponding to Area0,...,Area3. Each bucket is divided into 4 areas corresponding to R0,...,R3. This partitioning of the bucket is aimed for the parallelization of the sieving part.

For DNA matrix, we can use four 32-bit input buffers (`LDB`) and output buffers (`STD`). Each input and output buffers can accessed to the SDRAM in parallel. Then, the following procedures are executed by the four parallel processing.

Assume each factor base $(p, a, \log(p))$ stored in SDRAM as inputs of bucket sort part. Each pairs $(a, \log(p))$, named "packets", of sieving points calculated from the first sieving points and p, and log value $\log(p)$ are classified by using bucket sort algorithm. Each packet is stored in one of Bucket0,...,Bucket3

according to the sieving number (0,...,3) and RAM to be effected. The next section shows a concrete sieving algorithm with packets stored in buckets.

Sieving part

In the inside of DNA Matrix, 32 pieces of 16KByte RAM element exist. We assign one byte to each sieve memory, then the log values of $32 \cdot 16 \cdot 2^{10} = 2^{19}$ sieve points can be calculated at the maximum. Each RAM element can perform only update of one value at once, however, if the RAM element physically differ, independent and parallel operations are possible.

This part is consisted by the procedure in four parallel computations as same as in bucket sort part. The sieve position sorted in each partition in the Bucket0 for each RAM0,...,RAM3 are loaded into the four input buffers of DNA matrix. Then the memory updates are performed in parallel.

Check part

In this part, after the sieving process for all factor base, we check whether each updated value in the memory $\mathtt{S}[a]$ is larger than some threshold value, and output the sieve points satisfied such condition. The threshold values are obtained by linear interpolation from a few number of the sample points calculated correctly in advance.

4 Conclusion

This extended abstract proposes an efficient implementation of the sieving step in the number field sieve method of integer factorization on a dynamic reconfigurable processor "DAPDNA-2", by combining the pipeline method for small primes and the bucket sorting method for larger primes.

We have a lot of problems to be discussed. First of all, the lattice sieving for the sieving step should be discussed immediately. We have not applied the large prime variations. In this case, how to distinguish real relations and pseudo relations would be a serious problem. In addition comparisons to sophisticated algorithms for the sieving step such as TWIRL [11] and Geiselmann-Steinwandt's device [5]. Furthermore, we are planning to implement (some) algorithms on DAPDNA-2. We hope we can report these results near in future.

References

1. Kazumaro Aoki and Hiroki Ueda, Sieving Using Bucket Sort, Advances in Cryptology ASIACRYPT 2004, Vol. 3329. of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 2004.
2. Daniel J. Bernstein. Circuits for integer factorization: a proposal. preprint, 2001.
3. J. Franke, et al. RSA-576. Email announcement, December 2003.

4. Willi Geiselmann and Rainer Steinwandt. A dedicated sieving hardware. In Yuliang Zheng, editor, Advances in Cryptology – PKC 2003, Vol. 2567 of Lecture Notes in Computer Science (LNCS), pp. 254–266, Springer-Verlag, 2003.
5. Willi Geiselmann and Rainer Steinwandt. Yet another sieving device In Tatsuaki Okamoto, editor, Topics in Cryptology – CT-RSA 2004, Vol. 2964 of Lecture Notes in Computer Science (LNCS), pp. 278-291, Springer-Verlag, 2004.
6. IPFlex, DAPDNA Architecture, 2004. (`http://www.ipflex.com/en/E1-products/dd2Arch.html`)
7. Arjen K. Lenstra and H.W. Lenstra, editors. The development of the number field sieve, Vol. 1554 of Lecture Notes in Mathematics (LNM), Springer-Verlag, 1993.
8. Arjen K. Lenstra and Adi Shamir. Analysis and optimization of the TWINKLE factoring device. In Bart Preneel, editor, Advances in Cryptology – EUROCRYPT 2000, Vol. 1807 of Lecture Notes in Computer Science (LNCS), pp. 35–52. Springer-Verlag, 2000.
9. Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of Bernstein's circuit. In Yuliang Zheng, editor, Advances in Cryptology – ASIACRYPT 2002, Vol. 2501 of Lecture Notes in Computer Science (LNCS), pp. 1–26, Springer-Verlag, 2002.
10. Adi Shamir. Factoring large numbers with the TWINKLE device (extended abstract). In Çetin Kaya Koç and Christof Paar, editors, Cryptographic Hardware and Embedded Systems – CHES 1999, Vol. 1717 of Lecture Notes in Computer Science (LNCS), pp. 2–12, Springer-Verlag, 1999.
11. Adi Shamir and Eran Tromer. Factoring large numbers with the TWIRL device. In Dan Boneh, editor, Advances in Cryptology – CRYPTO 2003, Vol. 2729 of Lecture Notes in Computer Science (LNCS), pp. 1–26, Springer-Verlag, 2003.