



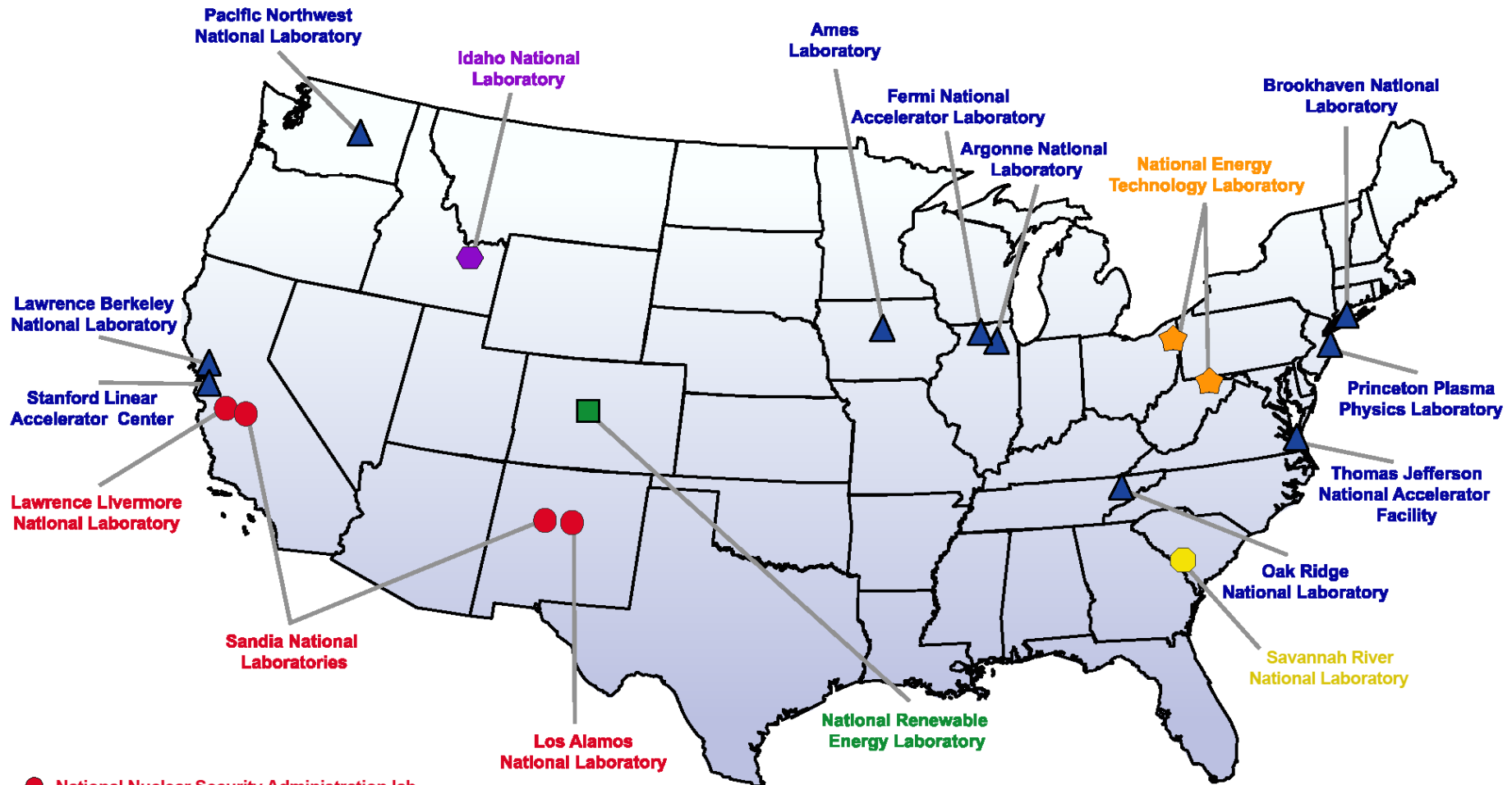
Efficient GCD Computation for Big Integers on Xeon Phi Coprocessor

*Scientific Computing Group
Jefferson National Lab
Newport News, VA 23606
U.S.A*

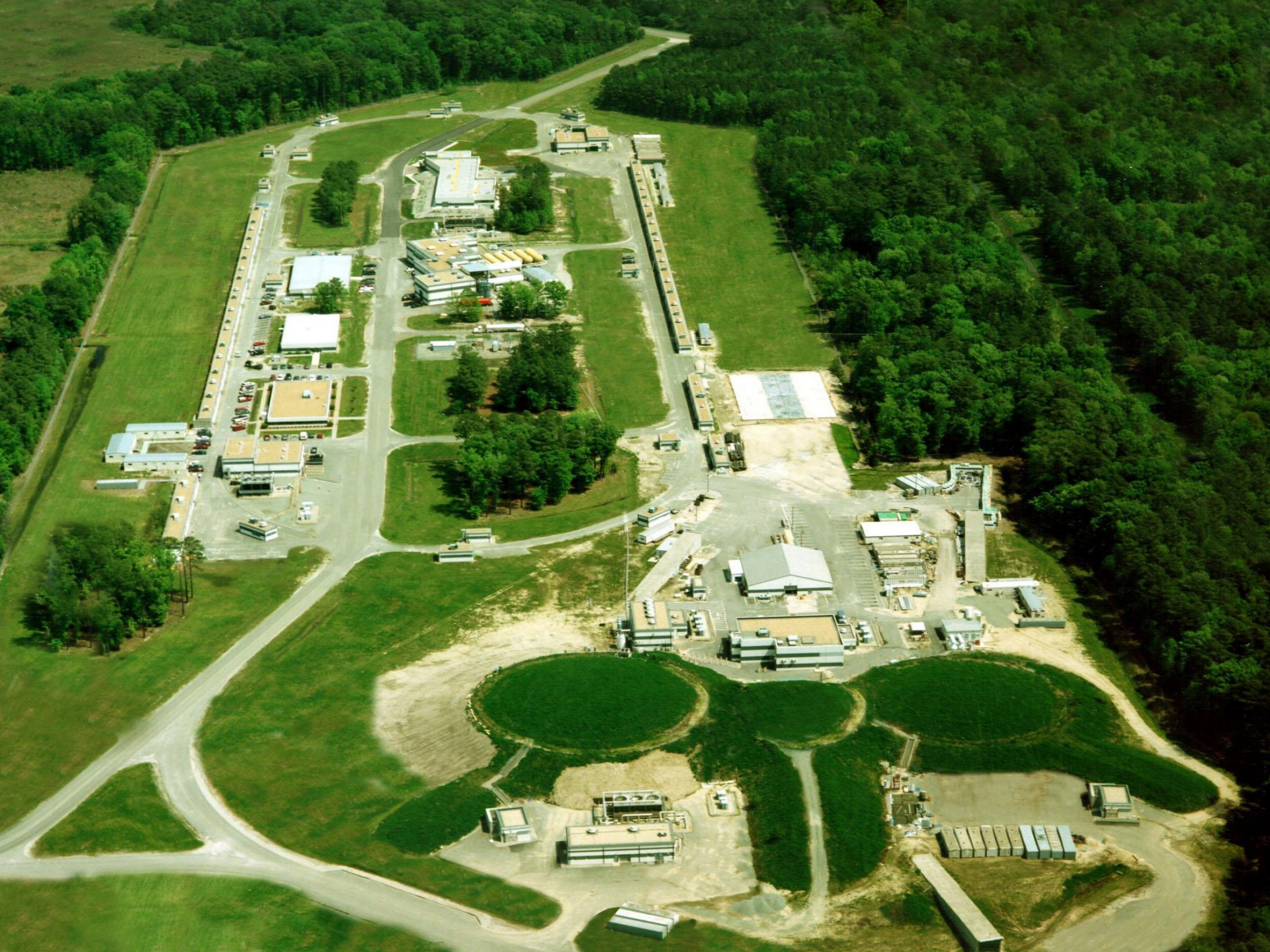
***Mayee F. Chen**
Williamsburg, VA 23188
U.S.A*



DEPARTMENT OF ENERGY NATIONAL LABORATORIES



- National Nuclear Security Administration lab
- Office of Energy Efficiency and Renewable Energy lab
- Office of Environmental Management lab
- ★ Office of Fossil Energy lab
- Office of Nuclear Energy, Science and Technology lab
- ▲ Office of Science lab



Organization

- Introduction
- Background
- Algorithms and Implementation
- Experimental Setup
- Performance Results
- Conclusions

Introduction

- Why are GCD calculations important?
 - RSA (ssh, ssl) public and private keys
 - public key: big integer n and e
 - n (1024 or 2048 bits) = $p * q$
 - » p and q are large prime numbers
 - $e = 2^{16} + 1 = 65537$
 - private key: big integer n and d
 - $d \cdot e \equiv 1 \pmod{(\phi(n))}$ where $\phi(n)=(p-1)(q-1) = n - (p + q - 1)$
 - Individual public key is considered secure
 - Difficult to factor n into $p * q$ if n is very large
 - Not so secure if there are many public keys to compare
 - $\text{GCD}(n_1, n_2) = p > 1$, $q_1 = n_1/p$, $q_2 = n_2/p$, then d_1, d_2 can be calculated
 - 4.7 million distinct 1024-bit RSA public keys collected, 12720 have a single large prime factor in common.

Ron was wrong, Whit is right

Arjen K. Lenstra¹, James P. Hughes²,
Maxime Augier¹, Joppe W. Bos¹, Thorsten Kleinjung¹, and Christophe Wachter¹

¹ EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland

² Self, Palo Alto, CA, USA

Abstract. We performed a sanity check of public keys collected on the web. Our main goal was to test the validity of the assumption that different random choices are made each time keys are generated. We found that the vast majority of public keys work as intended. A more disconcerting finding is that two out of every one thousand RSA moduli that we collected offer no security. Our conclusion is that the validity of the assumption is questionable and that generating keys in the real world for “multiple-secrets” cryptosystems such as RSA is significantly riskier than for “single-secret” ones such as ElGamal or (EC)DSA which are based on Diffie-Hellman.

Keywords: Sanity check, RSA, 99.8% security, ElGamal, DSA, ECDSA, (batch) factoring, discrete logarithm, Euclidean algorithm, seeding random number generators, K_9 .

Euclidean GCD Algorithm

- Euclidean GCD Algorithm

```
function gcd(a, b)
  if b = 0
    return a
  else
    return gcd(b, a mod
```

b)

mod operations are
expensive

```
function gcd(a, b)
  while b  $\neq$  0
    t := b
    b := a mod b
    a := t
  return a
```

Binary GCD Algorithm

- Binary GCD Algorithm

Input: integers of $x > y$

Output: The GCD value of x and y

repeat

if x is odd and y is odd then

if $x \geq y$ then

$\text{GCD}(x, y) = \text{GCD}((x - y)/2, y)$

else

$\text{GCD}(x, y) = \text{GCD}((y - x)/2, x)$

end if

else if x is odd and y is even then

$\text{GCD}(x, y) = \text{GCD}(y/2, x)$

else if x is even and y is odd then

$\text{GCD}(x, y) = \text{GCD}(x/2, y)$

else

$\text{GCD}(x, y) = 2 * \text{GCD}(x/2, y/2)$

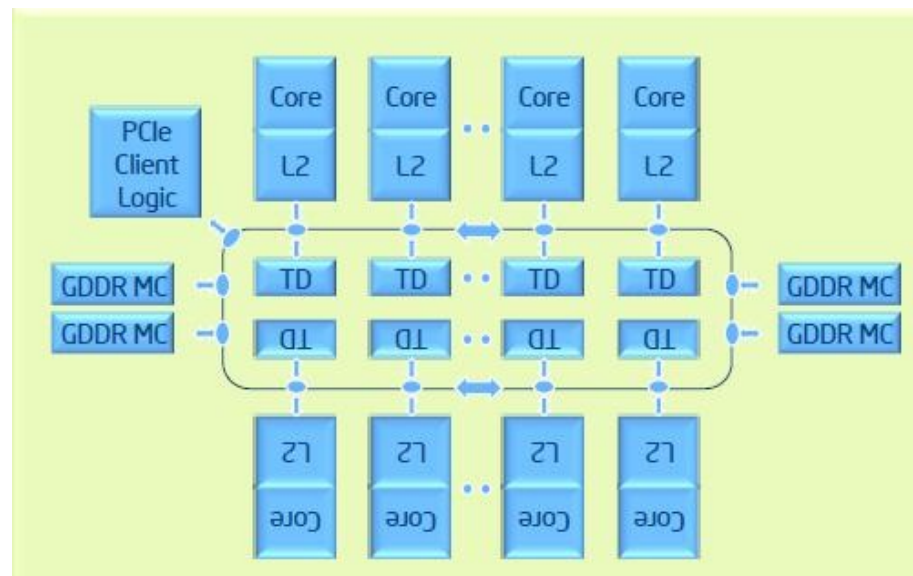
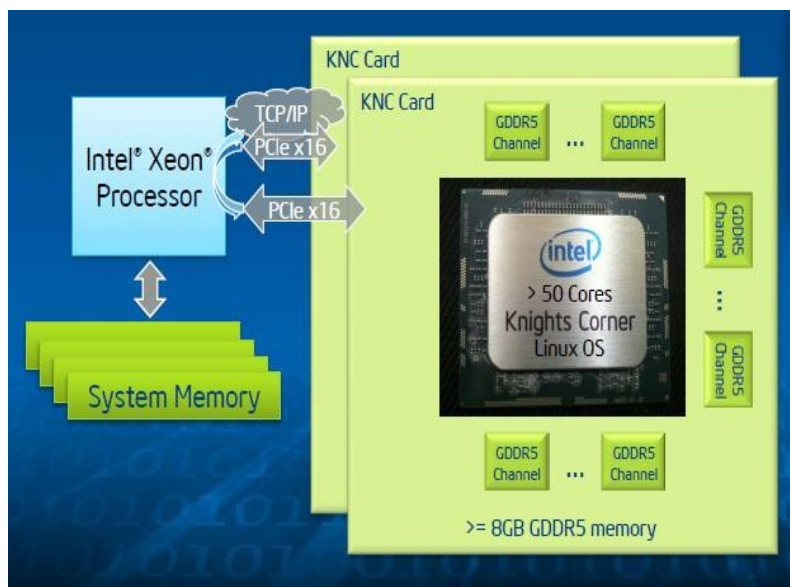
end if

until $\text{GCD}(x, y) = \text{GCD}(0, y) = y$;

Subtraction
Comparison
Left shift


1-bit right shift

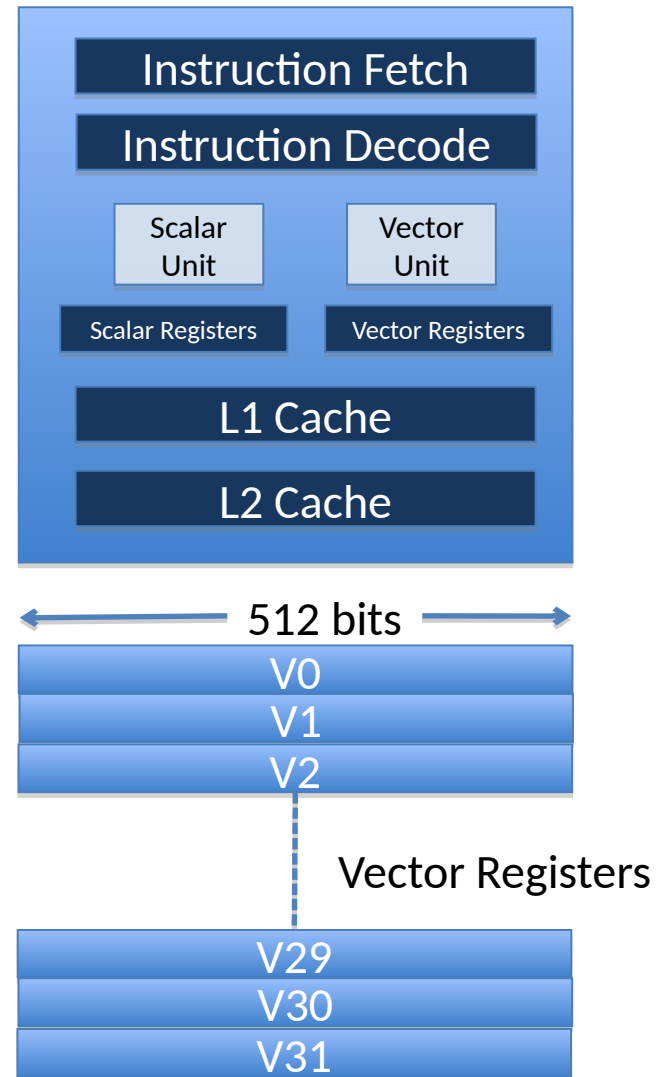
Intel Xeon Phi



4 hardware threads per core

Binary GCD on Xeon Phi

- Binary GCD is a sequential algorithm
 - Data dependent in the while loop
 - Auto-vectorization 
- Big Integers
 - Array of 32-bit integers
 - 2048 bits = 64 integers
- All integer arithmetic operations



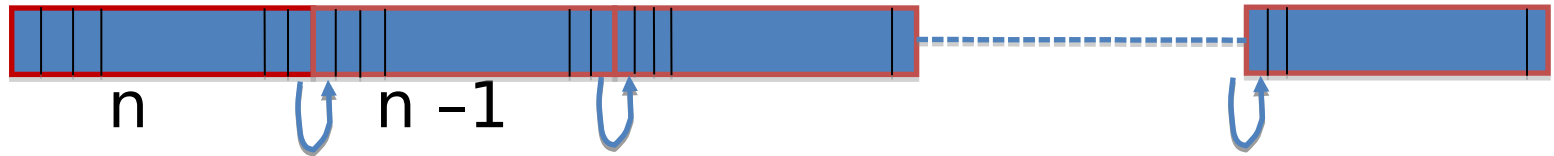
Parallel Binary GCD on Xeon Phi

- Increase throughput on many pairs of GCD calculations
 - Use 60 cores with 240 threads
 - openmp directives to utilize all threads

```
#pragma omp parallel for  
for (int i = 0; i < m; i++)  
    bn_gcd (c[i], a[i], b[i])
```

- Each thread carries out one pair of GCD calculation
 - Vectorization (using VPU)
 - 1-bit right shift
 - Subtraction
 - Comparison
 - Left shift

Right 1-bit Shift



- Sequential Algorithm

for each integer in the array a from $n-1$ to 0

a) get least significant bit (LSB) $c = (a[i] \& 1) << 31$



$a[i+1]$

$a[i]$

c

b) shift 1-bit right of $a[i]$;

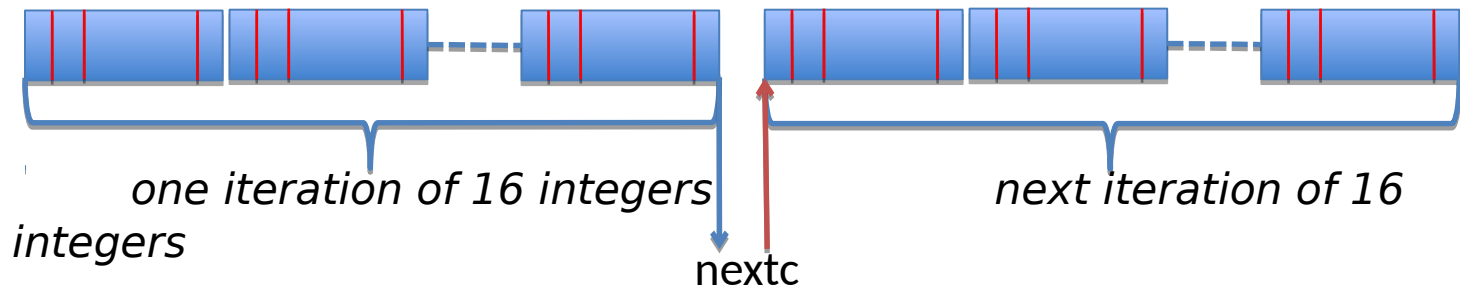


c) logic **or** with previous $(i+1)$ LSB value c ;



Parallel 1-bit Right Shift

- Parallel right 1-bit shift
 - Same ideas as the sequential algorithm
 - Doing 16 of 32-bit shifts at a time



– Challenges

- Number of iterations = size of big integer/16
- The size of integer array for a big integer may not have size of multiple 16.
 - Padding zeros
- Keep LSB values from one iterations to the next

Parallel 1-bit Right Shift

```
unsigned int prevc = 0, nextc;
```

```
for (i = n - 1; i >= 15; i -= 16) do
```

```
    m512i t ← vec_load(a[i], a[i - 1], ..., a[i - 15]);    _mm512_load_epi32
```

```
    m512i cbits ← vec((t and 1) << 31); {all LSBs}
```

```
    _mm512_and_epi32
```

```
    nextc ← cbits[0]; {LSB for next iteration}
```

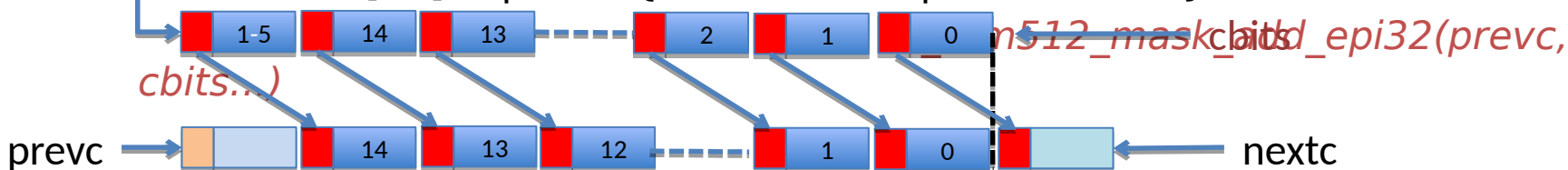
```
    _mm512_mask_reduce_add_epi32
```

```
    cbits ← vec shift element right(cbits, 1);
```

```
    _mm512_permutevar_epi32
```

```
    ↑ {move LSB from n to n - 1 element}
```

```
    cbits[15] ← prevc; {LSB from the prev iteration}
```



```
    t ← vec shift(t >> 1); {shift each integer}
```

```
    _mm512_srli_epi32(t, 1);
```

```
    t ← vec logic(t or cbits); {LSB from left integer}
```

```
    _mm512_or_epi32(t, cbits)
```

```
    prevc ← nextc;
```

```
    M ← vec store(t);
```

Subtraction

Algorithm 3 Parallel Subtraction

Input: big integer A, B: unsigned int a[n], b[n]

Output: big integer C: unsigned int c[n]

integer $br \leftarrow 0$, *nextbr*;

for $i = 0; i < n; i += 16$ **do**

$m512i$ $av \leftarrow vec_load(a[i], a[i + 1], \dots, a[i + 15]);$

$m512i$ $bv \leftarrow vec_load(b[i], b[i + 1], \dots, b[i + 15]);$

$m512i$ $brv \leftarrow vec_cmp_less(av, bv); \{\text{borrow flags}\}$

$nextbr \leftarrow brv[15]; \{\text{borrow flag for next iteration}\}$

$brv \leftarrow vec_shift_element_left(brv, 1);$

$\uparrow \{\text{move borrow flag from } n \text{ to } n + 1 \text{ element}\}$

$brv[0] \leftarrow br; \{\text{borrow flag from prev iteration}\}$

$m512i$ $cv \leftarrow vec_sub(av, bv); \{a - b\}$

$cv \leftarrow vec_sub(cv, brv); \{\text{subtract borrow flags}\}$

$br \leftarrow nextbr;$

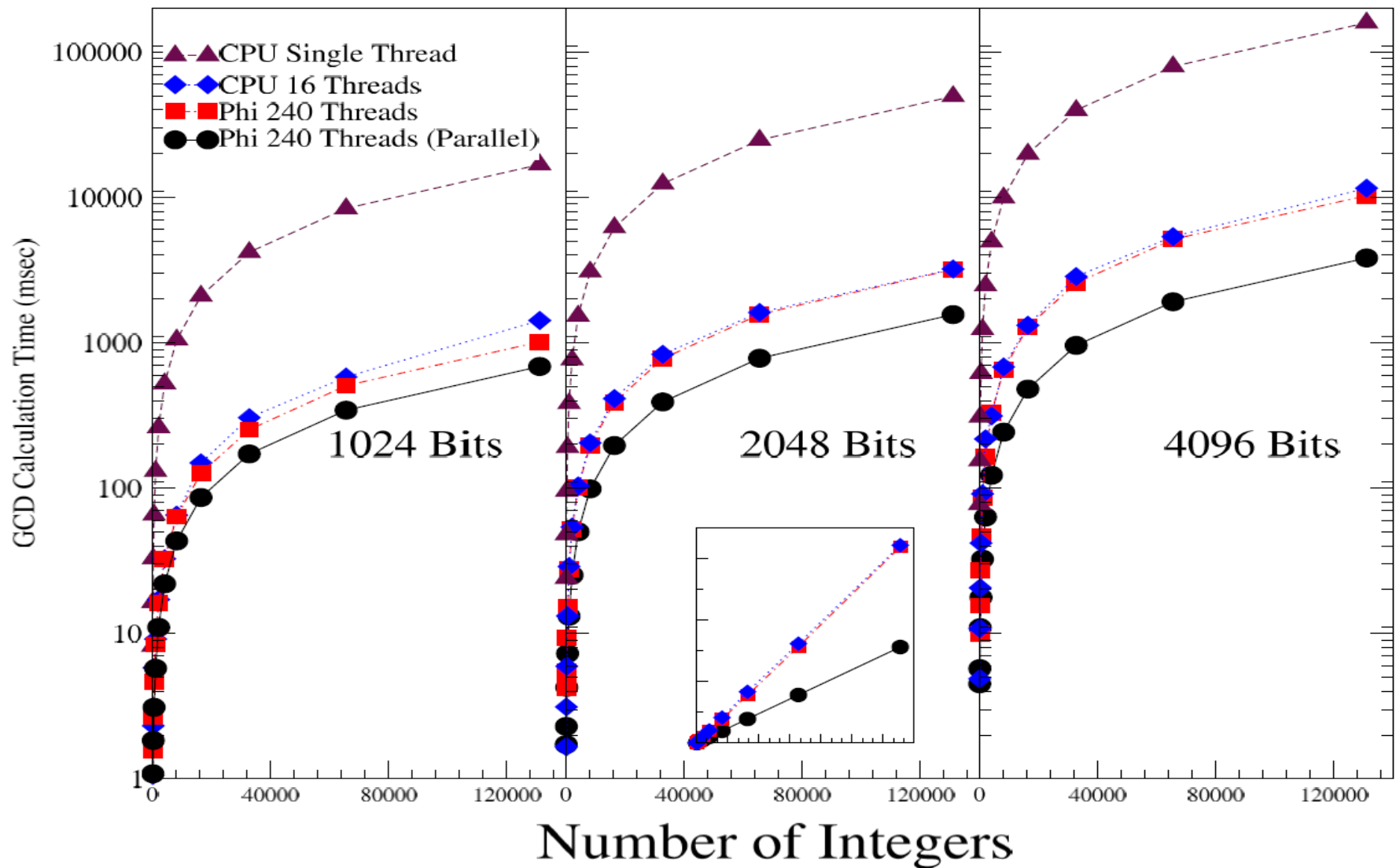
$c[i] \leftarrow vec_store(cv);$

end for

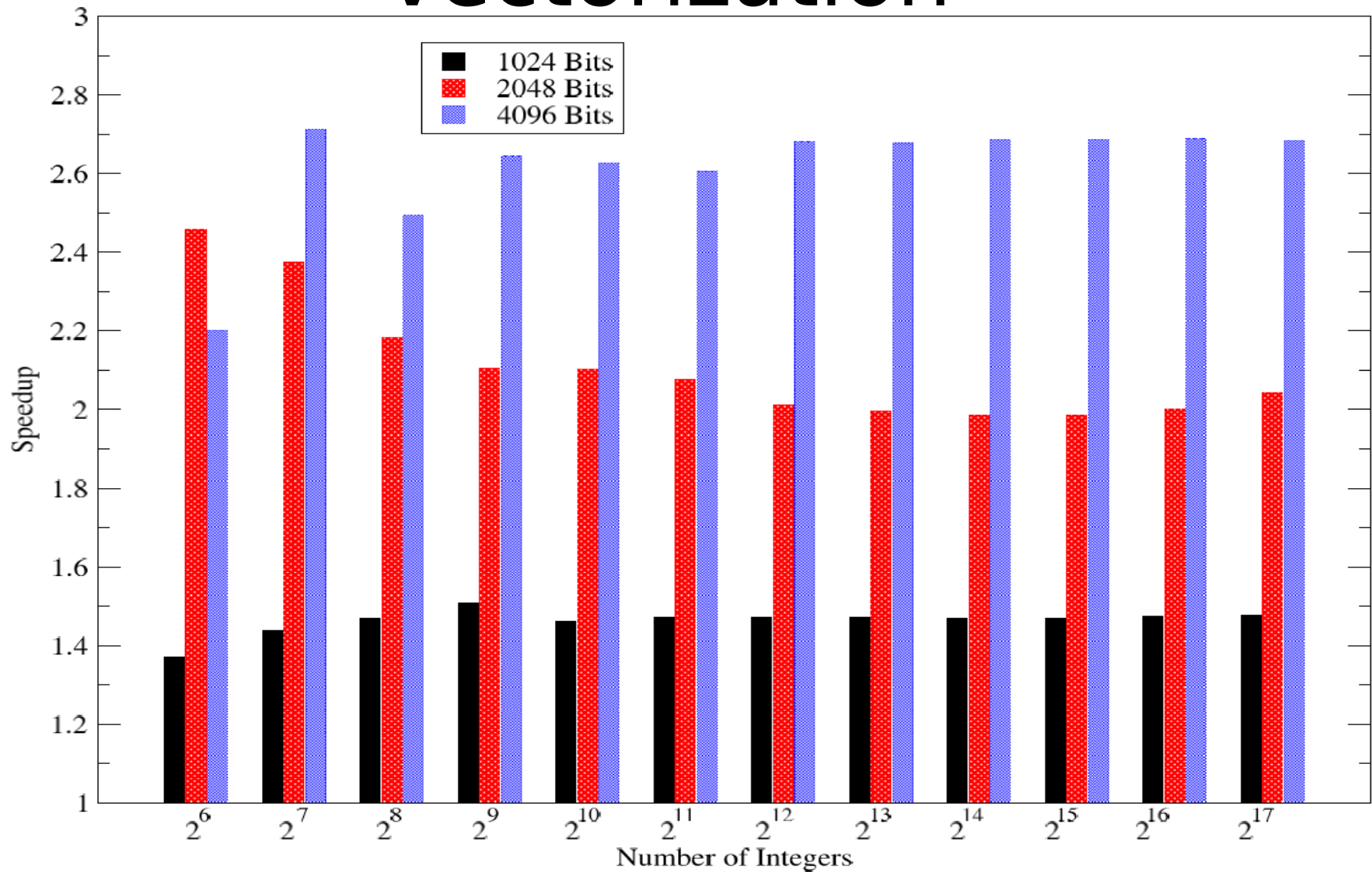
Experimental Setup

- 4 different settings
 - Sequential binary algorithm on a single CPU core
 - OpenMP utilizing 16 CPU cores
 - OpenMP utilizing 240 Xeon Phi Threads using non-vectorized algorithms
 - OpenMP utilizing 240 Xeon Phi Threads using vectorized algorithms
- Openssl-1.1-e
 - bn library
- Intel icc 13.0.
 - Compiler flag “-O3 -mmic” for Xeon Phi (Native Mode)
 - Compiler flag “-O3” for host CPUs
- Host
 - Dual 8-core Sandy Bridge CPUs (E5-2650 @2.00 GHz)
- Xeon Phi 7100P have 61 cores running at 1.238 GHz
 - Use 60 cores
- Randomly generated 2 arrays of big integers

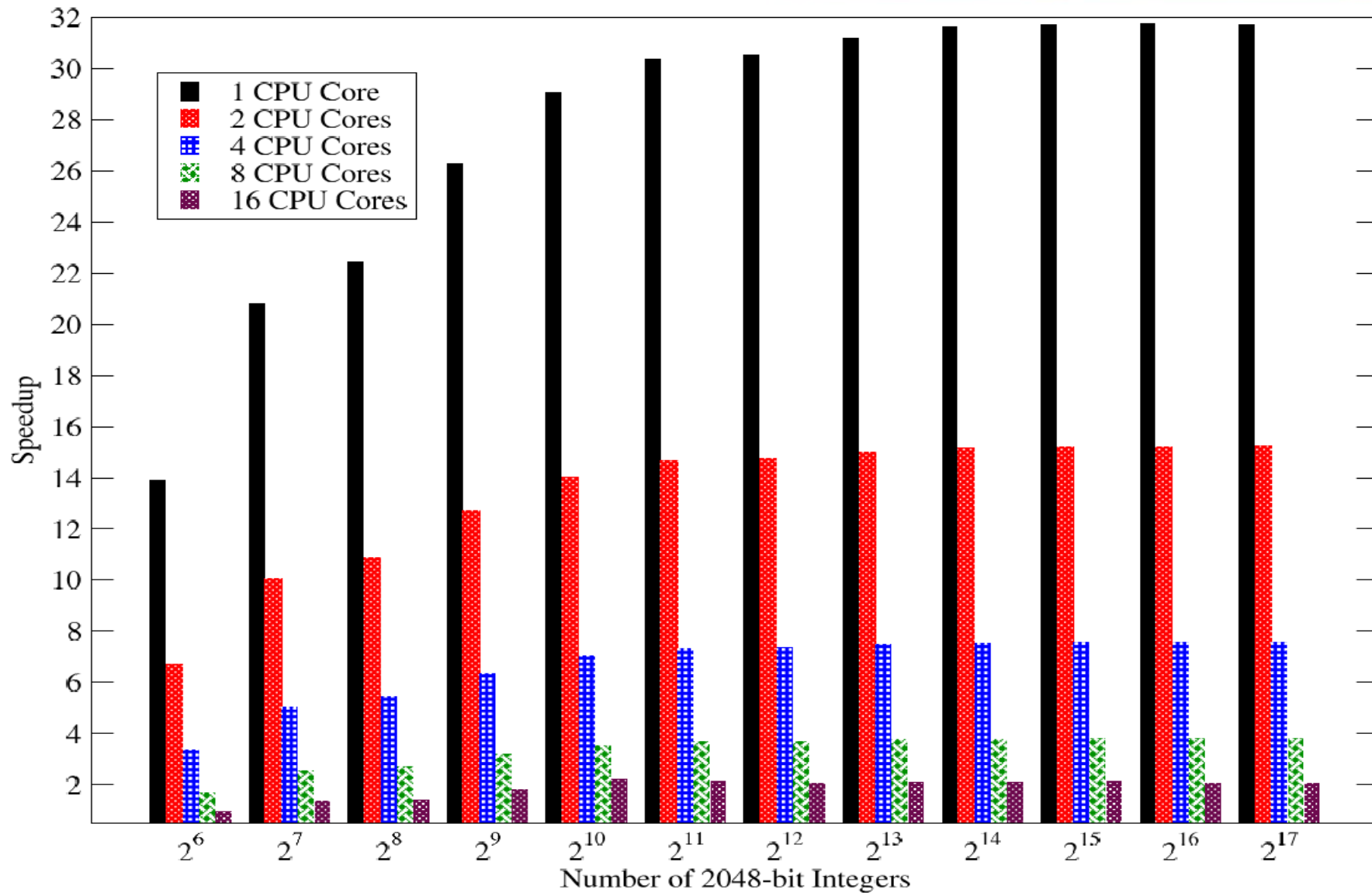
GCD Calculation Time



Speedups due to vectorization



Speedups over CPUs



Conclusions

- Two level parallelism applied to GCD calculations for big integers
 - OpenMP utilizing all cores
 - Vectorization utilizing VPU
 - Speedup of 30 over a single core CPU
 - Speedup of 2 over 16 CPU cores
 - Speedup of 2 over non-vectorized GCD on a Xeon Phi
- Xeon Phi is a viable high performance computing platform for integer arithmetic calculations as well as for floating point computations.