Programowanie równoległe i rozproszone

Projekt

Algorytm RSA z wykorzystaniem OpenMP i CUDA

Autorzy:
Kordian Kurdziel
Mateusz Maciejak

Wstęp

Zrównoleglanie obliczeń jest, przy obecnej architekturze wieloprocesorowych komputerów sprawą niezmiernie istotną. Rozdzielenie zadań na kilka współbieżnie przetwarzanych bloków pozwala na uzyskanie znacznego przyśpieszenia wszelkiego rodzaju obliczeń i operacji wymagających mocy obliczeniowej. Na rynku istnieje wiele technologii pozwalających na tworzenie oprogramowania działającego współbieżnie. Do tego celu wykorzystują różne podejścia. Message Passing Interface (MPI) wykorzystuje procesy procesora do podziału zadań, Open Multi-Processing (OpenMp) wykorzystuje wątki a technologia Compute Unified Device Architecture (CUDA) korzysta z ogromnej mocy obliczeniowej zawartej w kartach nVidia. W naszym projekcie w celu porównania technologii działających na różnym sprzęcie postanowiliśmy sprawdzić sposób i wydajność zrównoleglania obliczeń poprzez podział zadań między wątki procesora (OpenMp) lub wątki karty graficznej (CUDA). Dokument ten stanowi formę dokumentacji do projektu zaliczeniowego oraz zawiera wysnute przez nas wnioski i obserwacje na temat badanych technologii w przypadku szyfrowania algorytmem RSA.

1 Cel projektu

Celem projektu było zaimplementowanie algorytmu kryptograficznego jakim jest RSA z wykorzystaniem dwóch technologii OpenMP oraz CUDA. Jako punkt obserwacji został wybrany proces enkrypcji wiadomości, który jest długotrwały i w teorii powinien pozwolić się dobrze zrównoleglić. Dzięki zastosowaniu wyżej wymienionych technik zrównoleglania, oczekujemy że czas szyfrowania wiadomości ulegnie znacznej poprawie i zaobserwujemy ten proces w naszych rezultatach.

2 Algorytm RSA

Jeden z pierwszych i obecnie najpopularniejszych asymetrycznych algorytmów kryptograficznych z kluczem publicznym, zaprojektowany w 1977 przez Rona Rivesta, Adi Shamira oraz Leonarda Adlemana. Pierwszy algorytm, który może być stosowany zarówno do szyfrowania jak i do podpisów cyfrowych. Bezpieczeństwo szyfrowania opiera się na trudności faktoryzacji dużych liczb złożonych. Jego nazwa pochodzi od pierwszych liter nazwisk jego twórców.

Algorytm RSA składa się z trzech podstawowych kroków:

1. Generacja klucza publicznego i tajnego. Klucz publiczny jest przekazywany wszystkim zainteresowanym i umożliwia zaszyfrowanie danych. Klucz tajny umożliwia rozszyfrowanie danych zakodowanych kluczem publicznym. Jest trzymany w ścisłej tajemnicy. 2. Użytkownik po otrzymaniu klucza publicznego, np. poprzez sieć Internet, koduje za jego pomocą swoje dane i przesyła je w postaci szyfru RSA do adresata dysponującego kluczem tajnym, np. do banku, firmy komercyjnej, tajnych służb. Klucz publiczny nie musi być chroniony, ponieważ nie umożliwia on rozszyfrowania informacji – proces szyfrowania nie jest odwracalny przy pomocy tego klucza. Zatem nie ma potrzeby jego ochrony i może on być powierzany wszystkim zainteresowanym bez ryzyka złamania kodu.

3. Adresat po otrzymaniu zaszyfrowanej wiadomości rozszyfrowuje ją za pomocą klucza tajnego.

Szyfrowanie odbywa się za pomocą klucza publicznego (n, e). Wiadomość musi zo-

stać podzielona na części, a następnie każda część powinna zostać zmieniona na liczbę (która musi być większa od 0 i mniejsza niż n). W praktyce dzieli się wiadomość na fragmenty, z których każdy składa się z określonej ilości bitów.

Następnie każda liczba wchodząca w skład wiadomości jest podnoszona modulo n do potęgi równej e: $c_i = m_i^e \pmod{n}$

Algorytmu RSA można użyć wielokrotnie (przy zastosowaniu różnych kluczy) do zaszyfrowania danej wiadomości, a następnie odszyfrować ją w dowolnej kolejności. Otrzymany wynik będzie zawsze taki sam, bez względu na kolejność operacji. Jednakże, nie należy szyfrować w ten sposób wiadomości więcej niż dwa razy, ponieważ ujawniają się wtedy podatności na ataki oparte na chińskim twierdzeniu o resztach.

Szyfrowanie może zostać przeprowadzone również przy użyciu klucza prywatnego. Cała procedura jest identyczna do opisanej powyżej, z różnicą, że do szyfrowania trzeba będzie użyć klucza prywatnego (n, d). Natomiast odbiorca wiadomości będzie musiał użyć odpowiadającego mu klucza publicznego, w celu odszyfrowania wiadomości.

3 Metody zrównoleglenia algorytmu

3.1 MPI - OpenMP - Open Multi-Processing

Wieloplatformowy interfejs programowania aplikacji (API) umożliwiający tworzenie programów komputerowych dla systemów wieloprocesorowych z pamięcią dzieloną. Może być wykorzystywany w językach programowania C, C++ i Fortran na wielu architekturach, m.in. Unix i Microsoft Windows. Składa się ze zbioru dyrektyw kompilatora, bibliotek oraz zmiennych środowiskowych mających wpływ na sposób wykonywania się programu.

Dzięki temu, że standard OpenMP został uzgodniony przez głównych producentów sprzętu i oprogramowania komputerowego, charakteryzuje się on przenośnością, skalowalnością, elastycznością i prostotą użycia. Dlatego może być stosowany do tworzenia aplikacji równoległych dla różnych platform, od komputerów klasy PC po superkomputery.

OpenMP można stosować do tworzenia aplikacji równoległych działających na wieloprocesorowych węzłach klastrów komputerowych. W tym przypadku stosuje się rozwiązanie hybrydowe, w którym programy są uruchamiane na klastrach komputerowych pod kontrolą alternatywnego interfejsu MPI, natomiast do zrównoleglenia pracy węzłów klastrów wykorzystuje się OpenMP. Alternatywny sposób polegał na zastosowaniu specjalnych rozszerzeń OpenMP dla systemów pozbawionych pamięci współdzielonej (np. Cluster OpenMP Intela).

3.2 CUDA - Compute Unified Device Architecture

Opracowana przez firmę Nvidia uniwersalna architektura procesorów wielordzeniowych (głównie kart graficznych) umożliwiająca wykorzystanie ich mocy obliczeniowej do rozwiązywania ogólnych problemów numerycznych w sposób wydajniejszy niż w tradycyjnych, sekwencyjnych procesorach ogólnego zastosowania.

Integralną częścią architektury CUDA jest oparte na języku programowania C środowisko programistyczne wysokiego poziomu, w którego skład wchodzą m.in. specjalny kompilator (nvcc), debugger (cuda-gdb, który jest rozszerzoną wersją debuggera gdb umożliwiającą śledzenie zarówno kodu wykonywanego na CPU, jak i na karcie graficznej), profiler oraz interfejs programowania aplikacji. Dostępne są również biblioteki, które można wykorzystać w językach Python, Fortran, Java oraz Matlab. Pierwsze wydanie środowiska współpracowało z systemami operacyjnymi Windows oraz Linux. Od wersji 2.0 działa również z Mac OS X.

4 Opis programów

W celu zaobserwowania przyśpieszenia i zbadania czasu potrzebnego do zaszyfrowania wiadomości w badanych przypadkach, zostały zaimplementowane dwa programy realizujące ten sam cel. Większość kodu związanego ściśle z algorytmem pozostaje taka sama, jednak różnice w obu technologiach i inne procesy kompilacji zmusiły nas do stworzenia dwóch zupełnie oddzielnych programów z powtarzającą się logiką. Wszystkie procesy, ważniejsze funkcje w programach oraz kody źródłowe zostały opisane w poniższej sekcji.

Kod prezentujący funkcję enkrypcji wiadomości:

```
int encrypt(long int* in, long int exp, long int mod, long int* out, size_t
    len)
{
  for (int i=0; i < len; i++)
  {
     long int c = in[i];
     for (int z=1;z<exp;z++)
     {
        c *= in[i];
        c %= mod;
     }
     out[i] = c;
  }
  out[len]='\0';
  return 0;
}
```

Powyższy kod źródłowy zawiera wspólną dla obu technologii implementację szyfrowania wiadomości. Funkcja jako parametry przyjmuje wiadomość przekształconą na ciąg liczb typu long, referencje do dwóch zmiennych przechowujących wynik przekształceń podczas generowania klucza publicznego, wskaźnik do bufora przechowującego długość wyjściowej wiadomości oraz parametr określający długość wiadomości przeznaczonej do zaszyfrowania.

Funkcje obliczające klucz publiczny i prywatny prezentują się następująco:

```
int publickey(long int p, long int q, long int *exp, long int *mod)
{

  *mod = (p-1)*(q-1);
  *exp = (int)sqrt(*mod);
  while (1!=gcd(*exp,*mod))
  {
     (*exp)++;
  }
  *mod = p*q;
  return 0;
}

int privatekey(long int p, long int q, long int pubexp, long int *exp, long
    int *mod)
{
  *mod = (p-1)*(q-1);
  *exp = 1;
  long int tmp=pubexp;
  while(1!=tmp%*mod)
```

```
20   {
21       tmp+=pubexp;
22       tmp%=*mod;
23       (*exp)++;
24   }
25   *mod = p*q;
26   return 0;
27 }
```

W kolejnych linijkach postaramy się przedstawić cechy charakterystyczne dla obu testowanych technologii do zrównoleglania obliczeń.

4.1 OpenMP

Technologia OpenMp zyskała wielu zwolenników ze względu na bardzo niski koszt wprowadzenia jej do już istniejącego kodu. Aby zrównoleglić pętlę, wystarczy dodać przed nią odpowiednie klauzule charakterystyczne dla tej technologii. Na poniższym fragmencie kodu można zobaczyć jak w nieznacznym stopniu uległ zmianie kod funkcji szyfrowania.

```
1  int encrypt(long int* in, long int exp, long int mod, long int* out, size_t
       len)
2  {
3    #pragma omp parallel for schedule(dynamic) num_threads(numberOfThreads)
4    for (int i=0; i < len; i++)
5    {
6        long int c = in[i];
7        for (int z=1;z<exp;z++)
8        {
9            c *= in[i];
10           c %= mod;
11       }
12       out[i] = c;
13   }
14   out[len]='\0';
15   return 0;
16 }
```

4.2 Cuda

Technologia CUDA jest z pewnością dużo bardziej inwazyjna w już istniejący kod napisany przez programistę. Wymaga dodania specjalnych słów kluczowych przy każdej z metod, które mają być wykonane na karcie graficznej. Konieczne jest również kontrolowanie przez programistę, jakie zmienne są przechowywane w pamięci działającej na hoście a jakie na karcie graficznej. Z pewnością przejście z już istniejącego kodu do takiego, który można uruchomić współbieżnie na kartach z rodziny nVidia wymaga dużego narzutu na zmiany. W zamian technologia CUDA oferuje, bardzo często dużo większe przyśpieszenie obliczeń niż jest to możliwe na powszechnie wykorzystywanych procesorach.

```
1  int encrypt(long int* in, long int exp, long int mod, long int* out, size_t
       len)
2  //Encrypt an array of long ints
3  //exp and mod should be the public key pair
4  //Each number, c', is decrypted by
5  // c = (c'^exp)%mod
6  {
```

```
  long int *d_inout;
  //Allocate memory in the separate memory space of the GPU
  cudaMalloc(&d_inout, sizeof(long int)*len);

  //copy data to the GPU
  cudaMemcpy(d_inout, in, sizeof(long int)*len, cudaMemcpyHostToDevice); //
      copy to GPU

   float time;
   cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

  //Launch the kernel on the GPU with 1024 threads arranged in blocks of
      size BLOCKWID
  decrypt_kernel<<<BLOCK_SIZE, GRID_SIZE>>> (d_inout, exp, mod, len);

   //End timer and put result into time variable
   cudaDeviceSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);

   printf("%.4f\n", time);

  //copy data back from GPU
  cudaMemcpy(out, d_inout, sizeof(long int)*len, cudaMemcpyDeviceToHost);
      //copy from GPU

  out[len]=0; //Terminate with a zero
  cudaFree(d_inout);
  return 0;
}
```

Powyższy fragment przedstawia funkcję wywoływaną z głównej metody. Jak możemy zauważyć, programista musi zaalokować pamięć na urządzeniu z wykorzystaniem oferowanych przez technologię CUDA funkcji. Przedstawione wyżej metody są dostępne tylko i wyłącznie wtedy gdy program uruchamiany jest na sprzęcie wyposażonym w kartę nVidia, co stanowi spore ograniczenie i zmusza do przemyślanego definiowania takich funkcji. Jak możemy zauważyć oryginalny kod algorytmu szyfrowania musiał ulec znaczącym zmianom by mógł być uruchomiony na karcie graficznej. Logika szyfrowania została rozbita na funkcję uruchamianą na karcie graficznej. Specjalna klauzula **global** informuje kompilator, że funkcja jest tak zwanym kernelem i zostanie uruchomiona na urządzeniu. Wywoływana jest wykorzystując specjalną sementykę technologii CUDA. W poniższych fragmentach prezentujemy kod uruchamiany na karcie graficznej.

```
__global__ void decrypt_kernel(long int* inout, long int exp, long int mod,
    size_t len)
{
  for (int t = threadIdx.x + blockIdx.x*blockDim.x; t<len ;t+=blockDim.x*
      gridDim.x)
  {
     if (t<len) inout[t] = fastexp(inout[t], exp, mod);
  }

#ifdef __CUDA
```

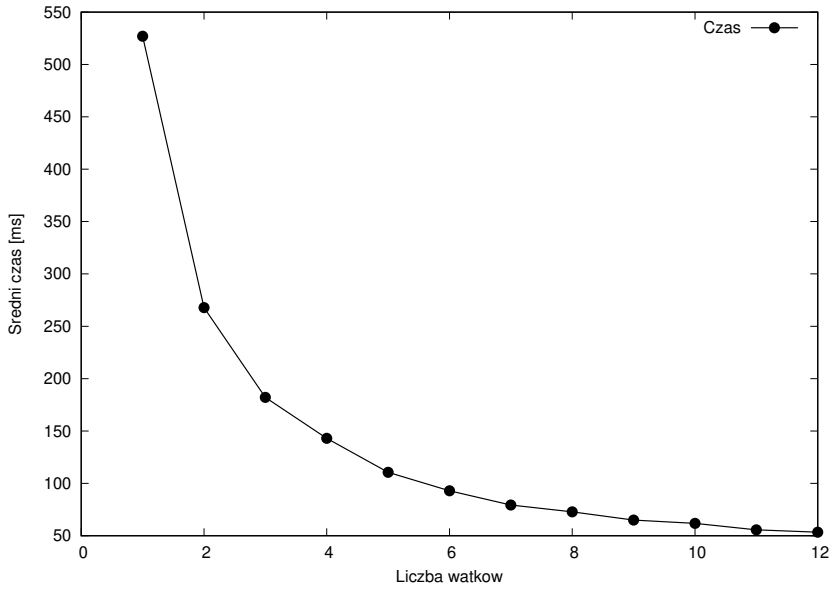
```
9  __device__ __host__
10 #endif
11 long int fastexp(long int base, long int exp, long int mod)
12 {
13   long int out = 1;
14   while(exp>0)
15   {
16       if(1==exp%2)
17       {
18           out*=base;
19           out%=mod;
20       }
21       base=base*base;
22       base%=mod;
23       exp/=2;
24   }
25   return out;
26 }
```

Klauzula **device** oznacza, że funkcja jest wywoływana z karty graficznej a nie z programu głównego. Dodatkowo funkcja ta też została opatrzona klauzulą **host** gdyż w przypadku braku karty nVidia w komputerze na którym uruchamiany jest problem prezentowana funkcja uruchamiana jest w tradycyjny sposób z wykorzystaniem mocy obliczeniowej jednostki CPU.

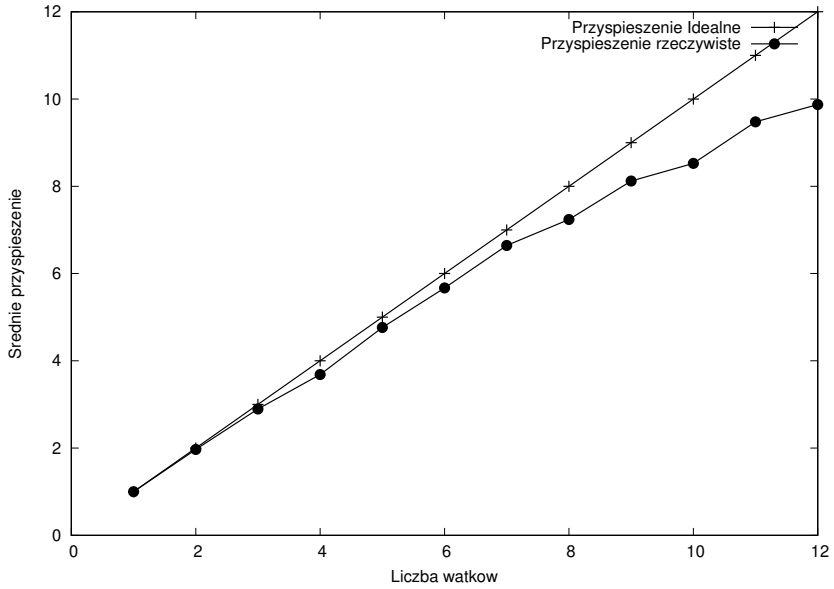
5 Pomiary

5.1 OpenMP

Poniższe wykresy i tabela z wynikami, przedstawiające zależność czasową oraz przyspieszenia zostały oparte na średnich wynikach programu uruchamianych lokalnie na maszynie wirtualnej. Wykorzystany został 6 rdzeniowy procesor Intel z technologią Hyperthreadingu.



Rysunek 1: Wykres zależności czasu wykonywania obliczeń od liczby wątków



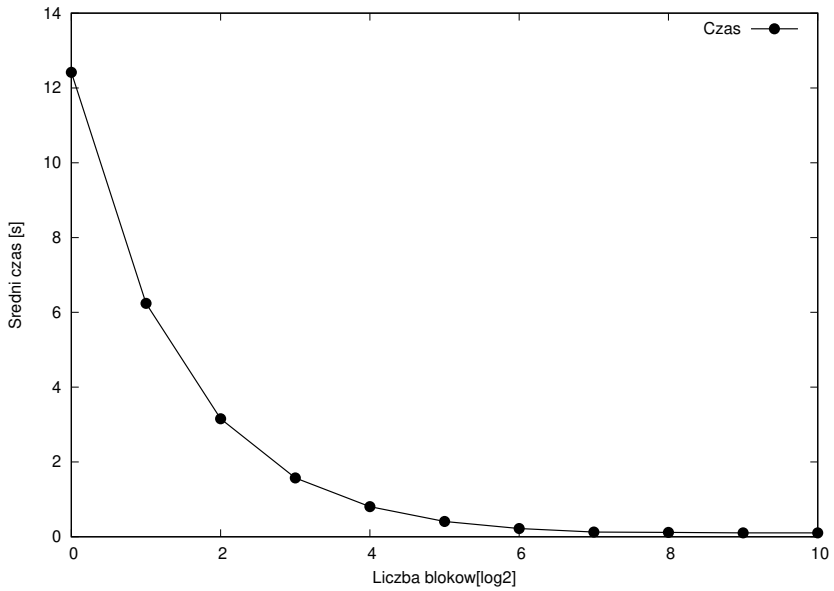
Rysunek 2: Wykres przyspieszenia działania programu w zależności od liczby wątków

Liczba wątków OpenMP	Czas[ms]	Przyspieszenie
1	526.98	1.00
2	267.85	1.96
3	182.06	2.89
4	143.019	3.68
5	110.60	4.76
6	92.95	5.66
7	79.34	6.64
8	72.80	7.23
9	64.90	8.11
10	61.82	8.52
11	55.61	9.47
12	53.37	9.87

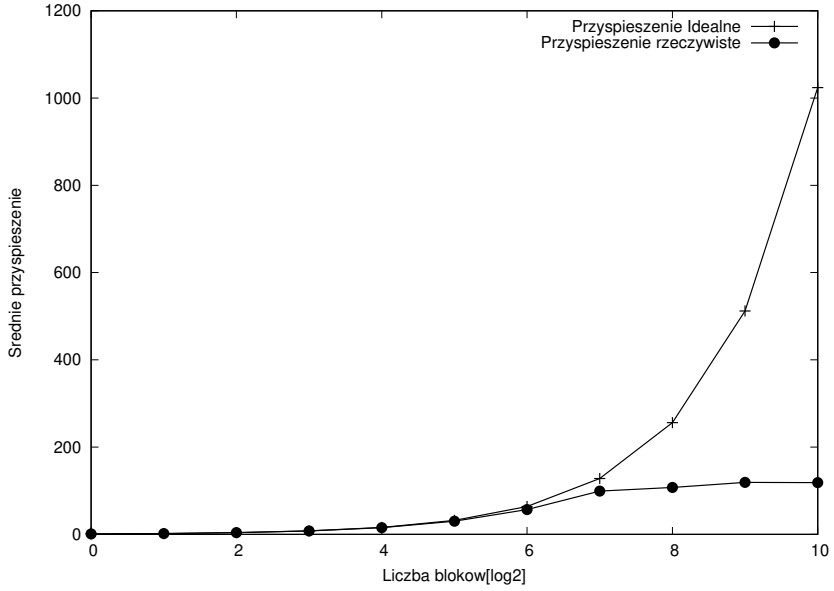
Jak widać dzięki zastosowaniu technologii OpenMP, wykorzystując dostępne wątki, udało się znacząco przyspieszyć wykonywanie programu. Uzyskane rzeczywiste przyspieszenie jest bliskie idealnemu, co może świadczyć o tym, że powyższa klasa problemów nadaje się całkiem dobrze do wykorzystywania obliczeń wielowątkowych. Jak można zauważyć na wykresie przyspieszenie działania programu wzrastało wraz z liczbą wykorzystywanych wątków nawet po przekroczeniu granicy jednostek fizycznych. Potem wraz ze wzrostem liczby wątków program dalej wykazywał przyspieszenie. Można więc wyciągnąć wniosek, że technologia Hyperthreadingu, wykorzystywana w procesorach Intela, działa aż tak wydajnie przy takich obliczeniach.

5.2 CUDA

Poniższe wykresy i tabela przedstawia wyniki uzyskane na serwerze CUDA. Czas obliczeń dla każdej ilości bloków przedstawionej na wykresie został przetestowany 5 razy a z wyników wyciągnięto średnią arytmetyczną.



Rysunek 3: Wykres zależności czasu wykonywania obliczeń od liczby bloków



Rysunek 4: Wykres zależności przyspieszenia od liczby bloków

Liczba bloków CUDA	Czas[s]	Przyspieszenie
1	12.41	1.00
2	6.24	1.99
4	3.15	3.93
8	1.57	7.89
16	.80	15.44
32	.41	30.27
64	.21	56.79
128	.12	99.23
256	.11	107.52
512	.10	119.26
1024	.10	118.67

Jak można zauważyć, czas obliczeń znacząco się zmniejsza wraz z rozpoczęciem zwiększania liczby bloków. Wraz z wzrostem liczby bloków różnice pomiędzy sąsiednimi czasami stają się coraz mniejsze.

6 Wnioski