ISSN 1975-8359(Print) / ISSN 2287-4364(Online)
The Transactions of the Korean Institute of Electrical Engineers Vol. 66, No. 1, pp. 100~106, 2017
http://doi.org/10.5370/KIEE.2017.66.1.100

FPGA 상에서 OpenCL을 이용한 병렬 문자열 매칭 구현과 최적화 방향

Parallel String Matching and Optimization Using OpenCL on FPGA

윤 진 명* · 최 강 일** · 김 현 진†
(Jin Myung Yoon · Kang-Il Choi · Hyun Jin Kim)

Abstract - In this paper, we propose a parallel optimization method of Aho-Corasick (AC) algorithm and Parallel Failureless Aho-Corasick (PFAC) algorithm using Open Computing Language (OpenCL) on Field Programmable Gate Array (FPGA). The low throughput of string matching engine causes the performance degradation of network process. Recently, many researchers have studied the string matching engine using parallel computing. FPGA's vendors offer a parallel computing platform using OpenCL. In this paper, we apply the AC and PFAC algorithm on DE1-SoC board with Cyclone V FPGA, where the optimization that considers FPGA architecture is performed. Experiments are performed considering global id, local id, local memory, and loop unrolling optimizations using PFAC algorithm. The performance improvement using loop unrolling is 129 times greater than AC algorithm that not adopt loop unrolling. The performance improvements using loop unrolling are 1.1, 0.2, and 1.5 times greater than those using global id, local id, and local memory optimizations mentioned above.

Key Words : GPU, PFAC, Parallel computing, String matching, OpenCL, Memory hierarchy

1. 서 론

현대 네트워크 통신에 있어서 malware와 spyware를 막는 것은 중요하다. 이에 따라 네트워크 상의 사용자 데이터에 악의적 공격 패턴이 있는지 검출할 수 있는 시스템으로 Network Intrusion Detection System(NIDS)이 있다. NIDS의 한 예인 Snort는 AC 알고리즘[1]과 밀접한 연관이 있다. Snort의 성능은 선택된 옵션의 알고리즘과 CPU의 병렬화 등의 하드웨어에 영향을 받는다[2][3].

최근에는 복잡한 명령어 처리에 중점을 둔 CPU 대신 수백 개의 코어로 구성되어 있는 Single Instruction Multiple Data (SIMD) 개념의 GPU가 데이터 분석에 많이 사용되고 있다. 문자열 매칭 분야에서는 악성 패턴을 GPU를 이용하여 검출하는 연구가 진행되고 있다. 그중 하나인 PFAC는 기존의 AC 알고리즘에서 Failure 천이를 제거하여 오토마타를 구성하여 속도를 향상시켰다[4].

웹의 발달과 High-performance computing (HPC)를 이용한 빠른 수행 속도, 기술 발전에 대한 빠른 대응 등의 장점으로 클라우드 컴퓨팅이 대두되고 이용되고 있다. 이에 따라 여러 유저의 정보가 데이터 센터에 저장되고, 이 경우 외부 공격에 대한 보안이 중요하다[5]. HPC 데이터 센터 유지에는 전력과 냉각 비용이 중요한 이슈이다. 이런 측면에서 FPGA는 기존의 GPU에 비해서 전력소비 대비 실행 속도에서 이점을 보인다[6][7]. FPGA 벤더는 OpenCL로 작성된 커널에 대해 병렬화 처리를 FPGA에서 수행할 수 있는 SDK를 제공하고 있다.

본 논문에서는 기존의 AC 알고리즘과 PFAC 알고리즘을 OpenCL을 이용하여 DE1-SoC 보드에서 구현하고 구동 속도를 분석한다. 테스트에서 실제 악성 패킷을 포함하고 있는 DEFCON [8]을 입력 스트림으로 사용하고, Snort 룰 셋 (Rule Set)을 오토마타로 구성하는 패턴으로 사용하였다. 테스트 케이스는 1개의 work-item을 이용한 AC 알고리즘과 PFAC를 이용한 global id를 사용한 경우, local id를 사용한 경우, local memory를 사용한 경우, loop unrolling을 적용한 경우를 비교 분석하여 최적화 방법을 모색하였다. 실험 결과로 PFAC 알고리즘에 loop unrolling을 적용한 경우가 최적화 하지 않은 AC 알고리즘보다 129배 그리고 PFAC에 대해 loop unrolling을 사용한 경우가 global id보다 1.1배, local id보다 0.2배, local memory 보다 1.5배의 성능 향상이 있었다.

† Corresponding Author : School of Electronics and Electrical Engineering, Dankook University, Yongin-si, Korea.
E-mail: hyunjin2.kim@gmail.com
* School of Electronics and Electrical Engineering, Dankook University, Yongin-si, Korea.
** Advanced Communications Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Korea.
Received : August 30, 2016; Accepted : November 14, 2016

2. 배경 지식

2.1 Aho-Corasick 알고리즘

AC 알고리즘은 Deterministic Finite Automata (DFA)를 구성하여 다중 패턴에 대해서 문자열 매칭을 수행한다. n개의 문자 입력이 이루어질 경우 AC 알고리즘은 O(n)의 시간이 걸리고 최악의 경우와 최상의 경우가 동일한 deterministic한 성능을 보인다.

그림 1은 “he,” “she,” “his,” “hers”의 4개의 패턴을 이용하여 DFA를 구성한 것이다. 각 원안의 수는 state의 수를 나타내고, state 0으로부터 각 state의 거리를 depth로 나타낸다. 예를 들어, state 0은 depth 0을 나타내고, state 1과 state 3은 depth 1을 나타낸다. 원으로 표현된 각 state 중 굵은 선으로 표현된 것은 그 상태에 도달한 경우 해당 패턴이 매칭 되었음을 의미한다. state 간의 이동을 나타내는 화살표에서 실선으로 표현된 것은 goto 천이이고, 점선은 failure 천이이다. DFA를 구성할 때, state의 공통된 접두사는 state를 공유한다. Failure 천이의 구성은 현재 state보다 낮은 depth의 state 중 가장 접미사를 많이 공유하는 state로 연결된다. 입력 스트림에 의해서 state 간 이동을 하게 되고, 이를 통해 DFA를 순회한다. 처음에는 state 0에서부터 시작한다. 현재의 state에서 goto 천이의 존재가 있는지 확인하고 없는 경우 failure 천이를 통해 이동하게 된다. 그림 1에서 나타나지 않은 state의 failure 천이는 state 0으로 이동하는 failure 천이로 시각적 편이를 위해 보이지 않고 있다.

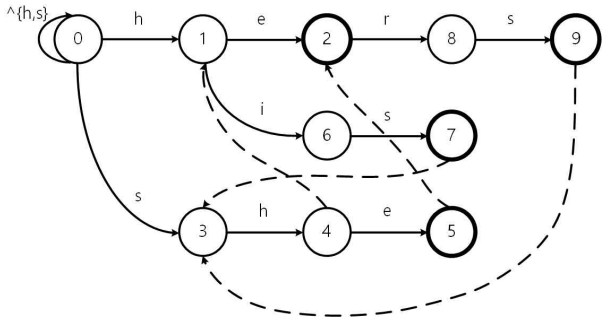

그림 1 AC 기반의 DFA.
Fig. 1 AC DFA.

예를 들어 입력 스트림으로 “sheis”라는 5개의 문자가 입력되었을 때, DFA는 state 0에서부터 순회를 시작한다. “s”에 대해서 state 3으로 이동하고 마찬가지로 다음 두 문자인 “he”의 입력으로 state 5로 이동한다. 이때, 굵은 원은 패턴을 찾은 경우이므로 “she”라는 패턴을 찾은 것을 알 수 있다. 다음 입력인 “i”에 대해서 다음 state로 이동이 없기 때문에 점선의 failure 천이를 통해서 현재 state까지의 패턴에서 가장 긴 접미사를 공유하는 state 2로 이동한다. 이 이동을 통해서 “he”라는 패턴을 찾은 것을 알 수 있고, 입력 “is”의해서 state 3으로 이동하여 순회를 끝낸다.

2.2 PFAC 알고리즘

PFAC 알고리즘[4]은 AC 알고리즘을 수정하여 GPU의 많은 코어에서 동작할 수 있다. PFAC 상태 머신은 AC DFA에서 failure 천이를 모두 제거하고, state 0의 “h,” “s”를 제외한 문자에 대해서 존재하는 goto 천이 역시 제거된다. 입력 스트림에 대해서 각 위치마다 GPU의 스레드 한 개씩을 할당하여 PFAC 상태 머신을 순회하게 되고, 유효한 goto 천이가 존재하지 않는다면 해당 스레드는 종료된다. 따라서 입력 스트림의 크기가 GPU가 한 번에 동작시킬 수 있는 스레드의 개수 이하라면 시간 복잡도의 최상의 경우는 O(1)이고, 최악의 경우는 O(m)이다. 여기서 m은 패턴 중 가장 긴 패턴의 길이를 나타낸다.

그림 2는 “he,” “she,” “his,” “hers”의 패턴을 이용하여 PFAC 상태 머신을 구성한 것이다. 스레드의 종료 조건은 유효한 goto 천이가 존재하지 않는 경우로, 입력 스트림이 “hei”인 경우를 예로 들 수 있다. “hei”의 경우 “h”에 대해서 state 0에서 state 1로 이동한다. “e”에 대해서 state 2로 이동하여 패턴 “he”를 찾고 “i”에 대해서 현재 state에서 다음 state로 이동 가능한 천이가 존재하지 않으므로 해당 스레드는 종료한다.

그림 3은 PFAC 알고리즘의 병렬 동작을 나타낸다. 각각의 상태 머신 위의 t_0, t_1 등으로 표기된 것은 스레드 번호를 나타낸다. 입력 스트림 “heisheihers”에 대해서 총 11개의 각각의 입력 스트림의 위치에서 시작하는 스레드가 할당된다. t_0은 첫 문자인 “h”부터 시작한다. “hei”까지 탐색을 하고 패턴 “he”를 찾은 후 종료한다. t_1의 경우 “e”부터 시작하고 유효한 천이가 존재하지 않으므로 종료한다. 각 스레드는 t_0이 “he,” t_3이 “she,” t_4가 “he,” t_7이 “hers”의 총 4개의 패턴을 찾는다.

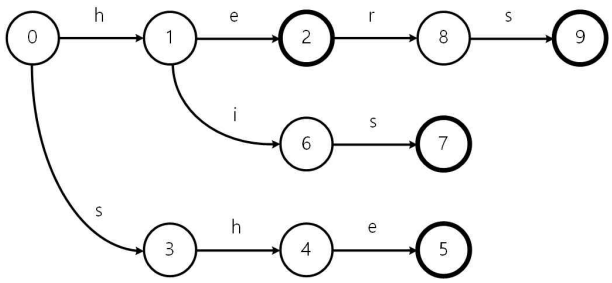

그림 2 PFAC 상태 머신.
Fig. 2 PFAC state machine.

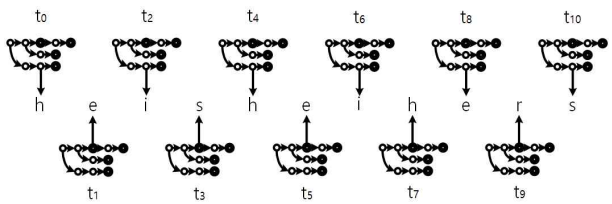

그림 3 PFAC 병렬 동작.
Fig. 3 PFAC parallel execution.

3. Altera OpenCL SDK에서의 메모리 계층구조

병렬 컴퓨팅을 이용한 문자열 매칭에서 가장 중요한 것 중 하나는 메모리 사용이다. OpenCL에서는 GPU의 스레드에 대응하는 work-item을 생성한다. 이것들이 SIMD로 동작하게 되고 동시에 메모리에 접근한다. 이로부터 병목 현상이 발생하여 병렬 컴퓨팅에서 성능을 저하시키는 요인 중 하나가 된다. FPGA에서는 메모리 계층 구조가 있고 이것을 사용하는 것은 병목 현상을 해결할 수 있다.

그림 4는 커널에서 사용하는 메모리의 계층구조이다. 각각의 회색 사각형은 메모리를 나타낸다. OpenCL에서 커널 프로그램을 수행하는 단위는 work-item이고, work-item의 모임은 work-group이다. work-item은 해당 work-item에서만 접근 가능한 private 메모리가 있고, work-group에서만 공유되는 local 메모리를 가지게 된다. 그리고 work-group이 공유하는 global/constant 메모리가 외부에 위치한다. Local 메모리는 global 메모리에 비해서 적은 지연시간 (latency)을 보이지만 작은 가용공간을 가진다.

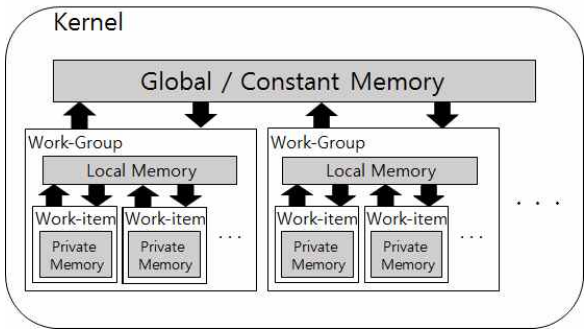

그림 4 커널의 메모리 계층 구조.
Fig. 4 Kernel's memory hierarchy.

4. DE1-SoC에서 OpenCL 최적화

4.1 Local id와 local 메모리 사용

그림 5는 PFAC 코드를 FPGA의 global id에 적용시킨 코드이다. 코드에서 global 메모리를 사용하는 변수는 입력 스트림(d_input_string)과 상태머신(d_PFAC_table), 매칭 결과 저장(d_match_result)이다. 9번째 줄의 get_global_id(0)에서 work-group의 id를 얻어온다. 이 코드에서는 입력 스트림의 문자 수만큼 work-group을 생성하고 각각의 work-group은 1개의 work-item을 가진다. 상태 머신의 순회는 while 문 안에서 이루어진다. 입력 스트림과 현재의 상태를 이용하여 다음 상태를 가져온다. 다음 상태가 유효한 상태이거나 매칭이 이루어지지 않았다면 계속해서 while loop를 돌며 상태 머신을 순회한다. 순회를 하며 다음 상태가 유효한 상태가 아니라면 (TRAP_STATE), 해당

```
1  __kernel void global_id_kernel
2                  (__global unsigned const * restrict d_PFAC_table,
3                   __global int const * restrict d_input_string,
4                   const int input_size,
5                   const int initial_state,
6                   const int num_finalState,
7                   __global int * restrict d_match_result)
8  {
9    __private int g_index = get_global_id(0);
10   __global char *string = (__global char *)d_input_string;
11   __private int inputChar = 0;
12   __private unsigned state = initial_state;
13   __private int pos = g_index;
14
15   while(pos < input_size){
16       inputChar = (int)(string[pos]);
17       state = d_PFAC_table[state*256 + inputChar];
18       if(TRAP_STATE == state){
19           break;
20       }
21       if(state <= num_finalState){
22           d_match_result[g_index] = state;
23       }
24       pos++;
25   }
26 }
```

그림 5 Global id를 이용한 커널 함수.
Fig. 5 Global id kernel.

work-item은 종료하고, 매칭이 이루어진다면 (<=num_finalState) 결과를 저장한다. PFAC의 FSM을 구성할 때, 앞서 설명한 그림 2의 굵은 원의 상태 값을 1에서부터 총 패턴의 수만큼을 사용한다. 그리고 num_finalState의 값은 전체 패턴의 수이고 이 값보다 작거나 같다는 것은 패턴이 매칭 되었다는 것을 의미한다. local id를 사용하는 방법은 코드의 9번째 줄의 get_global_id(0)를 get_local_id(0)로 교체하고 입력 스트림의 문자 수 만큼 work-item을 가진 1개의 work-group을 생성한다. 이런 변화로 work-item들은 local memory를 공유할 수 있다.

Local memory를 이용한 방식은 그림 6과 같다. 17번째 줄은 global 메모리에 있는 상태 머신의 초기 상태인 256개 요소를 local memory로 복사한다. 18번째 줄은 복사가 모두 이루어진 후에 상태 머신 순회가 이루어져야 하므로, 256개 요소의 복사가 모두 일어날 때를 기다리는 부분이다. 순회가 일어나는 부분은 20번째 줄의 while문이다. 21번째에서 25번째까지는 초기 상태에서 다음 상태로의 이동을 담당하는 부분이고 27번째부터의 while문에서 이후의 상태 이동이 이루어진다. 실제 실험에서 상태 머신의 초기 상태를 local 메모리로 복사하는 부분과 local 메모리를 이용하여 초기 상태에서 다음 상태로 이동하는 부분은 빠른 속도를 보인다. 하지만 전체 효율은 global id를 이용한 방법보다 저하되는데, 그 이유는 1번째 줄에서 명시하였듯이 256개의 work-item만을 생성하였기 때문이다. DE1-SoC의 커널 프로그램에서 사용 가능한 local 메모리의 최대 크기는 2KB로, 초기 상태의 벡터를 저장하는 것만으로 1KB를 소비하고, work-item 자체에서 지역 변수로 사용하여 local 메모리에 저장되는 것들을 포함하면 256개보다 많은 work-item을 생성하기는 힘들다.

```
__attribute__((reqd_work_group_size(256,1,1)))
__kernel void local_memory_kernel
                        (__global unsigned const * restrict d_PFAC_table,
                         __local int * restrict d_PFAC_initial_table,
                         __global int const * restrict d_input_string,
                         const int input_size,
                         const int initial_state,
                         const int num_finalState,
                         __global int * restrict d_match_result)
{
  __private int l_index = get_local_id(0);
  __global char *string = (__global char *)d_input_string;
  __private int inputChar = 0;
  __private unsigned state = initial_state;
  __private int pos = l_index;

d_PFAC_initial_table[l_index] = d_PFAC_table[state*256 + l_index];
barrier(CLK_LOCAL_MEM_FENCE);

 while(pos < input_size){
      inputChar = (int)(string[pos]);
      state = d_PFAC_initial_table[inputChar];
      if(state <= num_finalState){
          d_match_result[l_index] = state;
      }
      if(TRAP_STATE != state){
          while(pos < input_size){
              pos++;
              inputChar = (int)(string[pos]);
              state = d_PFAC_table[state*256 + inputChar];
              if(TRAP_STATE == state){
                  break;
              }
              if(state <= num_finalState){
                  d_match_result[l_index] = state;
              }
          }
      }

      l_index += 256;
      pos = l_index;
  }
}
```

그림 6 Local memory를 이용한 커널 함수.
Fig. 6 Local memory kernel.

4.2 Loop unrolling

Loop unrolling은 FPGA와 GPU 모두에게 스루풋 향상을 이룰 수 있는 방법이다. FPGA에서는 회로의 여유가 존재하면 loop unrolling을 수행하는 것이 일반적으로 성능의 향상을 이룰 수 있다. OpenCL의 병렬 동작은 파이프라인으로 동작하고 위에서 기술한 방법의 loop는 loop unrolling보다 파이프라인 depth가 짧다. 따라서 현재 work-item이 동일한 하드웨어를 여러 번 점유하기 때문에 뒤에서 수행해야 할 work-item의 실행이 뒤로 밀릴 수가 있다.

문자열 매칭 알고리즘에서는 현재 상태와 현재의 입력 문자에 의하여 다음 상태가 결정된다. 그리고 이러한 동작의 연속이기 때문에 각 work-item마다 다음 상태가 존재하지 않을 때까지 loop를 이용하여 상태 머신을 순회하게 구현한다. 이와 같은 data dependency로 인하여 Altera OpenCL SDK에서 제공하는 #progma unroll 키워드를 이용하여 loop unrolling을 구현할 수 없다. 본 논문에서는 여러 테스트 경우 중 하나를 depth 0에 대해서 loop unrolling을 수행하였다. 그림 7은 #progma unroll 키

```
__kernel void loop_unroll_kernel
                        (__global unsigned const * restrict d_PFAC_table,
                         __global int const * restrict d_input_string,
                         const int input_size,
                         const int initial_state,
                         const int num_finalState,
                         __global int * restrict d_match_result)
{
  __private int l_index = get_local_id(0);
  __global char *string = (__global char *)d_input_string;
  __private int inputChar = 0;
  __private unsigned state = initial_state;
  __private int pos = l_index;

  inputChar = (int)(string[pos]);
  state = d_PFAC_table[state*256 + inputChar];
  if(state <= num_finalState){
      d_match_result[l_index] = state;
  }
  if(TRAP_STATE != state){
      while(pos < input_size){
          pos++;
          inputChar = (int)(string[pos]);
          state = d_PFAC_table[state*256 + inputChar];
          if(TRAP_STATE == state){
              break;
          }
          if(state <= num_finalState){
              d_match_result[l_index] = state;
          }
      }
  }
}
```

그림 7 Loop unrolling을 적용한 커널 함수.
Fig. 7 Kernel adapted loop unrolling.

워드를 사용하지 않고 loop unrolling을 구현한 커널 코드이다. 상태 머신의 초기 상태에서 다음 상태로의 이동을 15~19번째 줄과 같이 while loop 밖에서 동작시켜 depth 0에서의 상태 이동은 loop에서 제외한다. 앞서 설명한 local id와 마찬가지로 work-item을 입력 스트림의 수만큼 생성한다. 커널의 파이프라인 depth는 이런 work-item의 수보다 적다. 또한 depth 0에서 depth 1로의 이동을 수행하고 loop에서 동작을 수행하기 때문에 성능에 영향을 미치지 않을 수 있다는 예상을 할 수 있다. 그렇지만, 실제 NIDS에서 매칭은 잘 일어나지 않고 패턴의 다양성으로 depth를 깊이 들어갈수록 상태의 이동이 이루어질 가능성이 적다[9]. 이런 이유로 depth 0에 대해서 loop unrolling을 적용하는 것은 성능을 향상시킬 수 있다.

5. 실험 결과

본 논문에서는 1개의 work-item을 이용한 AC 알고리즘, global id, local id, local memory를 사용, depth 0에 대해서 loop unrolling을 적용한 5개의 테스트 경우에 대해서 성능을 분석하였다. global id와 local id는 각각 1개의 work-item을 가진 입력 스트림의 문자 수만큼의 work-group을 생성하고, 문자 수만큼의 work-item을 가진 1개의 work-group을 생성한다. 따라서 커널의 구현 방법은 동일하고, work-item의 id를 get_global_id(0) 혹은 get_local_id(0)을 이용하여 얻는 것의 차

이가 있다. local memory는 상태 머신의 초기 상태인 256개의 인자를 global 메모리에서 local 메모리로 복사한 후 문자열 매칭을 진행한다. DE1-SoC 보드의 local 메모리는 2KB로 초기 상태를 복사하면 50%를 차지한다. 각각의 work-item에서 사용하는 변수를 포함하면 72%로, 256개의 work-item을 생성하였다. Loop unrolling은 입력 스트림의 문자 수 만큼의 work-item을 가진 1개의 global-item을 생성하고 초기 상태에서 다음 상태로의 이동을 loop 밖에서 동작하여 depth 0에 대해서 loop unrolling을 적용하였다.

표 1 실험 대상 FPGA의 리소스.
Table 1 FPGA resource.

global 메모리	64MB
local 메모리	2KB
global 메모리 cache size	4KB
최대 work-item 할당 가능량	2,147,483,647
최대 동작 주파수	1,000MHz

FPGA의 리소스를 고려하여 입력 스트림으로 31M까지의 DEFCON 입력 스트림과 1,926개의 패턴까지를 실험에 이용하였다. 표 2와 표 3는 각각 31MB까지 DEFCON 크기에 패턴의 수를 변화시키며 스루풋을 측정한 것과, 927개의 패턴의 수에 입력 스트림의 변화에 따라 스루풋을 측정한 것이다. 가장 높은 성능을 보여주는 것은 loop unrolling을 적용하였을 경우이다. Loop unrolling의 평균 성능 향상은 global id보다 114%, local id 보다 28%, local memory보다 156%, AC 알고리즘 보다 12,931%의 향상을 보였다.

표 4를 보면 각 테스트 케이스에 따른 FPGA 자원 사용량을 알 수 있다. 문자열 매칭에서는 비교 연산이 주를 이루기 때문에 FPGA 내의 Digital Signal Processor (DSP) 블록들을 사용하지 않는다. Logic utilization의 경우, global id의 사용량이 가장 적고 local id는 global id보다 약간 늘어났다. 전체 실행 절차는 같지만 이와 같은 변화가 생기는 원인은 local 메모리의 공유를 위한 추가 회로가 생기는 것으로 추측할 수 있다. Global id와 local id의 차이는 후자는 local 메모리를 공유한다는 것이다. Local memory는 global 메모리에서 초기 상태를 local 메모리에 저장하는 회로와 local 메모리 사용량의 증가로 logic utilization과 memory blocks의 사용량이 증가한다. Loop unrolling은 초기 상태에서 다음 상태로의 이동 동작을 loop에서 밖으로 빼내어 동작하기 때문에 local memory보다는 아니지만 local id보다 logic utilization과 memory blocks의 사용량이 증가한다.

표 4 각 테스트 경우의 리소스 사용량.
Table 4 Resource usages of each test case.

리소스	global id	local id	local memory	loop unrolling
logic utilization	26%	29%	45%	40%
dedicated logic registers	11%	12%	20%	17%
memory blocks	20%	25%	72%	41%
DSP blocks	0%	0%	0%	0%

표 5는 global id와 local id의 메모리 stall, 점유율, bandwidth efficiency와 cache hits를 비교한 표이다. 이 표의

표 2 패턴 변화에 따른 스루풋.
Table 2 Throughputs of sets of patterns.

조건					커널 스루풋 (Gbps)				
패턴의 수	DEFCON 크기	매칭된 패턴 수	상태의 수	메모리	global id	local id	local memory	loop unrolling	AC 알고리즘
484	31M	58,734	7,452	7,452	0.292	0.517	0.206	0.589	0.004
972	31M	63,247	12,729	12,729	0.222	0.347	0.195	0.483	0.004
1,437	31M	63,247	16,132	16,132	0.272	0.495	0.198	0.558	0.004
1,926	31M	84,947	19,018	19,018	0.257	0.461	0.195	0.565	0.004

표 3 입력 스트림 변화에 따른 스루풋.
Table 3 Throughputs according to input streams.

조건				커널 스루풋 (Gbps)				
DEFCON 크기	패턴의 수	매칭된 패턴 수	상태의 수	global id	local id	local memory	loop unrolling	AC 알고리즘
8M	927	21,658	12,729	0.229	0.367	0.232	0.509	0.004
16M	927	42,210	12,729	0.225	0.357	0.209	0.499	0.004
24M	927	54,791	12,729	0.221	0.347	0.198	0.484	0.004
31M	927	84,947	12,729	0.222	0.347	0.195	0.483	0.004

수치는 local id가 global id에 비해서 높은 성능을 보이는 이유를 설명하고 있다. 상태 머신으로부터 읽어오는 값의 cache hits를 제외하면 local id의 성능이 더 뛰어난 것을 확인할 수 있다. Global id는 다수의 work-group이 global 메모리에 접근하여 병목 현상의 발생으로 stall이 높고 메모리 점유율과 대역폭의 효율성이 낮다. 또한 입력스트림의 cache hits 비율은 극단적으로 낮다. 문자열 매칭의 경우 잦은 메모리 접근을 요구하는 특성을 보이기 때문에 이와 같은 차이는 성능에 큰 영향을 보인다.

표 5 리소스 사용 비교.
Table 5 Comparisons in terms of resource usages.

	stall	occupancy	bandwidth (efficiency)	cache hits	
				입력 스트림	상태 머신
global id	39.60%	49.33%	17.07%	0%	59.68%
local id	5.75%	90.08%	48.29%	98.60%	59.63%

Local id에서 global 메모리에 접근하는 부분은 그림 5에서 16번째 줄의 입력 스트림을 읽어오는 부분과 17번째 줄의 상태 머신에서 다음 상태를 얻어오는 부분, 그리고 22번째 줄의 출력 상태를 저장하는 부분이다. Loop unrolling의 global 메모리에 접근하는 부분은 그림 7에서 15, 23번째 줄의 입력 스트림을 읽어오는 부분, 16, 24번째 줄의 상태 머신에서 다음 상태를 읽어오는 부분과 18, 29번째 줄의 매칭 결과를 저장하는 부분이다. 이와 같이 loop unrolling의 파이프라인 depth가 local id보다 깊고 실제 네트워크에서 문자열 매칭은 상태 이동이 많지 않기 때문에 앞서 동작하는 work-item에 의해서 뒤에 동작하는 work-item의 대기하는 현상이 적다.

6. 결 론

이 논문에서는 OpenCL과 PFAC를 이용한 FPGA 상에서 병렬 문자열 매칭에 대해서 성능을 분석하였다. 그와 더불어 여러 테스트 케이스에 대한 최적화 방법에 대해서 알아보았다. 문자열 매칭의 특성인 높은 빈도의 메모리 접근을 이유로 global id의 사용보다는 local id의 사용이 적합하고, 파이프라인 병렬화의 특성상 더 높은 성능을 얻기 위하여 파이프라인 depth를 늘려야 하고 이를 위하여 초기 상태에 대하여 loop unrolling을 적용하고 성능을 측정하였다. 따라서 OpenCL을 이용한 FPGA의 병렬 문자열 매칭은 local id를 이용하고 여분의 logic element들이 존재한다면 loop unrolling을 수행하는 것이 효율적이다. 이것은 PFAC를 적용한 다른 최적화 방법보다 0.2~1.5배의 성능향상이 있었다.

감사의 글

본 연구는 미래창조과학부 및 정보통신기술연구진흥센터의 정보통신·방송 연구개발사업의 일환으로 수행하였음.
[B0101-16-0233, 스마트 네트워킹 핵심 기술 개발]

References

[1] Aho, Alfred V., Margaret J. Corasick., "Efficient String Matching: An Aid to Bibliographic Search", Communications of the ACM, 18.6, pp. 333-340, 1975.
[2] Roesch, Martin. "Snort: Lightweight Intrusion Detection for Networks", LISA, Vol. 99, No. 1, pp. 229-238, 1999.
[3] Snort_Users Manual 2.9.8.2 Available: http://www. snort.org
[4] Lin, Cheng-Hung, et al. "Accelerating Pattern Matching Using a Novel Parallel Algorithm on Gpus", IEEE Transactions on Computers, 62.10, pp. 1906-1916, 2013.
[5] Cloud computing. Available: Wikipedia, 2016.5.1.
[6] Radar Processing: FPGAs or GPUs? Available: https://www.altera.com
[7] Chen, D., Singh, D., "Invited paper: Using OpenCL to Evaluate the Efficiency of CPUs, GPUs and FPGAs for Information Filtering", In 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 5-12, 2012.
[8] DEFCON, http://cctf.shmoo.com, 2013.
[9] Nishimura, T., Fukamachi, S., Shinohara, T., "Speed-up of Aho-Corasick Pattern Matching Machines by Rearranging States", In String Processing and Information Retrieval, 2001. SPIRE 2001. Proceedings. Eighth International Symposium on IEEE, pp. 175-185, 2001.

저 자 소 개

윤 진 명(JinMyung Yoon)
2014년 단국대 전자전기공학과 졸업.
2014년 9월~현재 : 동대학원 전자전기공학과 석사과정.

최 강 일(Kang-Il Choi)

1992년 : KAIST 전자계산학과 학사.
1994년 : 서강대학교 전자계산학과 석사.
2011년 9월~현재 : 한국전자통신연구원 스마트노드플랫폼연구실 선임연구원.

김 현 진(HyunJin Kim)

1997년 : 연세대학교 전기공학과 졸업.
2010년 : 연세대학교 전기 전자공학과 졸업 (박사).
2011년 9월~현재 : 단국대학교 전자전기공학과 조교수.