Journal of Institute of Control, Robotics and Systems (2013) 19(12):1072-1080
http://dx.doi.org/10.5302/J.ICROS.2013.13.9037 ISSN:1976-5622 eISSN:2233-4335

반도체 웨이퍼 고속 검사를 위한 GPU 기반 병렬처리 알고리즘

The GPU-based Parallel Processing Algorithm for Fast Inspection of Semiconductor Wafers

박 영 대, 김 준 식*, 주 효 남
(Youngdae Park[1], Joon Seek Kim[1,*], and Hyonam Joo[2])
[1]Dept. of Electronics Engineering, Hoseo University
[2]Dept. of Digital Display Engineering, Hoseo University

Abstract: In a the present day, many vision inspection techniques are used in productive industrial areas. In particular, in the semiconductor industry the vision inspection system for wafers is a very important system. Also, inspection techniques for semiconductor wafer production are required to ensure high precision and fast inspection. In order to achieve these objectives, parallel processing of the inspection algorithm is essentially needed. In this paper, we propose the GPU (Graphical Processing Unit)-based parallel processing algorithm for the fast inspection of semiconductor wafers. The proposed algorithm is implemented on GPU boards made by NVIDIA Company. The defect detection performance of the proposed algorithm implemented on the GPU is the same as if by a single CPU, but the execution time of the proposed method is about 210 times faster than the one with a single CPU.

Keywords: wafer inspection, machine vision, GPU, parallel processing, CUDA

I. 서론

오늘날 반도체 부품은 각종 제품의 개발과 함께 많은 양의 데이터 처리 분야와 데이터베이스 구축, 각종 제어분야, 컴퓨터 산업, 의료기기, 군사 무기 등 전방위적인 산업에 사용되고 있다. 이러한 전자제품들의 성능은 반도체 부품의 성능에 의해 결정된다고 할 수 있으며, 이들 반도체 부품은 하루가 다르게 고집적화, 소형화, 고성능화를 향해 발전해 가고 있다. 이에 따라 반도체 산업은 생산비용의 절감효과와 높은 수율을 목표로 제품의 생산 장비의 효율화와 제품의 완성단계에서의 검사뿐 아니라, 공정과 공정사이의 검사 등 생산 전체의 효율적인 관리 시스템은 물론 부품의 재료가 되는 웨이퍼(wafer)검사의 중요성도 점점 상승하였다. 이러한 반도체산업은 흔히 장비(FA: Factory Automation)산업이라 일컫는데 그만큼 장비가 투자적인 측면이나 기술적인 측면에서 큰 비중을 차지하고 있다는 것을 의미한다[1].

현재 반도체 소자의 집적도를 높이기 위해 다양한 연구가 진행되고 있으며, 특히 복수의 실리콘 웨이퍼를 수직으로 적층하는 웨이퍼 스텍기술에 대한 활발한 연구가 진행되고 있으나, 이러한 경우 웨이퍼간의 결합이 불완전하여 웨이퍼의 접합면사이의 에어갭(airgap)이 발생될 수 있을 뿐 아니라, 스텍 제작을 위한 다양한 공정을 거치는 동안 웨이퍼 후면에는 스크래치(scratch)또는 크랙(crack)등이 발생할 수 있다. 이러한 에어갭, 스크래치, 크랙 등은 웨이퍼의 스텍이나 웨이퍼후면을 정확히 검사할 수 있는 방법이 요구되며, 현재 300mm급의 웨이퍼에서 차후 450m 웨이퍼가 산업에 범용적으로 사용되어, 더 많은 회로를 집적 및 실장할 수 있는 기술들이 발전하고 있으나, 집적기술의 발전과 더불어 육안으로는 검사할 수 없는 nm급 결함들이 현재보다 더 많이 발생할 것이다. 이러한 nm급 결함들을 검사하기 위해 영상을 통한 검사기술의 발전과 자동검사공정으로 변화가 진행되고 있다[2].

현재 국내 반도체 회사의 웨이퍼 검사장비는 대부분 외산장비들로서 국산화가 미비한 수준으로 외산 검사장비의 검사 방법, 광학계 등은 전혀 알려진바 없다, 최근에는 결함의 유 · 무 판정뿐 아니라 검사기술을 통해 반도체 제조에 적용되는 많은 공정들을 관리하기 위한 많은 노력들이 이루어지 지고 있다, 고집적화 되는 반도체의 경우 고해상도 카메라를 이용해 적게는 30~40배, 많게는 100배 이상을 확대할 수 있는 광학기술을 이용한 검사방법이 필요한데, 현재 이러한 검사장비들의 검사시간은 수초에서 수십 초의 시간을 소비하게 되며, 반도체 생산업체의 가장 큰 문제라 할 수 있는 생산효율을 충족시킬 수 없는 상황이다. 이러한 문제를 해결하기 위해 알고리즘을 다중 스레드(multi-thread)로 분산처리 하는 기술이 도입되고 있으나, 다중 스레드간의 동기화 문제, 웨이퍼의 크기와 반비례 하여 결함의 크기는 점점 미세화 되고 이에 따라 취득하는 영상의 데이터 크기는 커지면서 분산처리 역시 한계에 다가가고 있다. 2007년 NIVIDIA社에서 발표한 CUDA (Compute Unufude Device Architecture)는 이러한 대용량 데이터 처리의 분산처

* Corresponding Author
Manuscript received August 20, 2013 / revised September 15, 2013 / accepted October 4, 2013
박영대, 김준식: 호서대학교 전자공학과
(heilowa@naver.com/joonskim@hoseo.edu)
주효남: 호서대학교 디지털디스플레이공학과(hnjoo@hoseo.edu)
※ 이 논문은 호서대학교의 재원으로 학술연구비지원을 받아 수행된 연구임.

리 방법에 새로운 방향을 제시한다. 더욱이 CUDA는 별도의 설치나 추가적인 장비의 부착 없이 G8X GPU로 구성한 GForce8 시리즈급 이상에서 제약 없이 동작한다[15]. 한 개의 스레드에서 수천개의 스레드까지 병렬로 처리가 가능한 CUDA는 그래픽처리를 위해서 사용되었던 GPU를 대용량 데이터 처리까지 가능하도록 만들었으며, C언어의 확장 형태로 제작되어져 접근성, 진입장벽이 낮을 뿐 아니라, 범용성을 갖기 위해 OpenCL, OpenGL, OpenCV(OpenCV는 2.3버전 이상부터 지원한다)등을 지원하며, 이미 영상처리외의 여러 분야에서 널리 사용되고 있다. 기존 범용적인 사용에 초점을 맞추어 생산되어지는 CPU에는 한정적으로 장착된 데이터 연산에 적합한 ALU (Arithmetic and Logic Unit) 다수를 병렬로 배치하여, 각각의 ALU 그룹을 제어하는 컨트롤러를 통해 GPU를 동작시킨다. 이를 통해 동일한 시간에 대용량 데이터처리가 가능하며, CPU와 비교해 적게는 수배에서 많게는 수백배까지 처리시간의 단축이 가능하다. 또한 CPU에서 문제가 되었던 스레드간의 동기화 문제를 각각의 컨트롤러의 제어를 통해 손쉽게 제어가 가능해지고, 데이터 처리를 위해 사용할 수 있는 메모리 역시 최고 6GB까지 사용가능 할뿐이니라, CPU에서 제한적으로 사용가능했던 캐시메모리가 개발자의 의지에 따라 사용이 가능해졌다.

현재 CPU와 GPU간의 데이터 전송 역시 본 논문에서 사용된 Tesla모델의 경우 반이중방식(half duplex)이 아닌 양방향통신(full duplex)을 통해 전송시간을 최소화 할 수 있다.

본 논문의 실험 대상은 300mm급 실리콘 웨이퍼로서 실제 공정에서 제작 및 사용되는 웨이퍼 패턴 영상의 결함을 검출하는 알고리즘이다. 반도체 웨이퍼 결함을 검출하는 알고리즘들은 정확성이 뛰어난 반면, 실제 생산 공정에 적용하기에 시간적인 요구조건을 충족하지 못하는 경우가 대부분이다. 하지만 본 논문에서는 실제 생산라인에서 사용될 수 있는 웨이퍼 검사 알고리즘을 CUDA를 이용하여 고속화 및 병렬처리에 최적화하도록 하는 방법을 제안하였다.

본 논문의 구성은 II 장에서는 웨이퍼 검사 개요 및 GPU에 관해 설명한다. III 장에서는 반도체 웨이퍼 검사 알고리즘에 대한 설명과 고속화에 대한 설명을 하고, IV 장에서는 고속화된 알고리즘을 병렬처리시 최적화하는 방법에 대한 설명하며, V 장에서는 본 논문에서 제안된 방법을 통한 실험과 그 결과에 대한 고찰을 진행한다. 마지막으로 VI 장에서 본 논문의 결론을 맺는다.

II. 웨이퍼 검사 개요 및 GPU

1. 웨이퍼 검사 개요

반도체 웨이퍼 검사란 현대의 반도체 산업의 핵심 재료인 반도체 웨이퍼를 검사하는 기술로서 스마트폰, 컴퓨터, 게임기 등과 같은 전자 제품에서 동작할 수 있도록 해주는 반도체 소자가 그 대상이다. 이 검사 기술은 반도체 생산업체의 매출과 생산수율에 지대한 영향을 미치는 공정으로서 오늘날 많은 웨이퍼 제조업체들의 가장 큰 관심사 중의 하나이다.

반도체 검사 기술은 이와 같은 반도체 웨이퍼를 대상으로 검사가 이루어지며, 이러한 반도체 검사 기술은 대부분 외산 장비에 의존하고 있으며, 국내에는 웨이퍼를 직접적으로 검사하는 기술이 많이 부족한 실정이다. 하지만 웨이퍼 패턴의 공정은 점점 미세화 되고, 수율 향상을 위해 보다 정밀하고 빠른 검사 장비를 요구하고 있는 상황에서 미세화된 공정은 결함의 종류가 더욱 다양해졌으며, 이를 정확하고 빠르게 검사하기 위해서는 2D검사의 검출력 향상 및 병렬처리 기법 등을 통한 검사 시스템 개발이 이루어져야 한다.

제조 공정중의 웨이퍼는 재료, 공정, 설계 등의 오류를 포함하여 생산되고, 이런 결함들을 검출하기 위해 특정 공정이 완료 후 검사를 진행한다. 결함은 일정한 형식 없이 무작위로 발생하는 경우와 일정한 형태를 갖고 반복적으로 발생하는 경우로 구분된다. 전자의 경우 공기 중의 미세 먼지나 공정중의 불순물로 인한 경우가 대부분이며 예측이 불가능하다. 후자의 경우 패턴 전달시 발생하는 경우가 많으며 조절이 가능하기 때문에 이러한 결함에 대한 검사에 많은 관심이 집중되고 있다[3].

웨이퍼 결함 검사 시스템은 웨이퍼 결함의 검사 방법에 따라 크게 두 가지로 분류할 수 있다. 육안으로 검출 가능한 웨이퍼 결함이나, 특정 크기 이상의 국부적인 결함을 검사하는 매크로 검사(macro inspection)는 검사속도가 비교적 빠르기 때문에 공정 중 검사(in-line inspection)가 가능하다. 반대로 육안으로 검사가 불가능하며, 50um 미만의 미세결함을 검출하는 마이크로 검사(micro inspection)는 웨이퍼당 검사 시간이 10분 이상이 소요 되며, 보통 웨이퍼 샘플을 활용한 공정 외 검사(off-line inspection)나 다른 방법으로 발견된 결함의 세부 분석에 사용된다.

2. 웨이퍼 결함 종류

일반적인 웨이퍼 결함의 유형은 크게 전역에 걸쳐 발생하는 결함과 국부적인 결함의 두부류로 나눌 수 있다. 이러한 웨이퍼 결함의 종류는 크게 패턴결함과 범프 결함으로 나눌 수 있다. 두 결함은 크기 및 특성이 상이하여, 서로 다른 광학계로 검사를 진행하며, 웨이퍼의 크기가 커지고 집적되는 회로의 정밀도가 고도화 되면서 패턴결함의 경우 100nm이하 급까지 볼 수 있는 광학계를 설계 및 설치하여 결함을 검출하고 있다.

패턴 결함의 경우 대부분 세정, 광역 평탄화(global planarization)공정 등의 문제로 발생하는 결함들로 원(circle), 나선(spiral), 선(line) 그리고 긁힘(scratch)결함이 일반적으로 발생하나, 그 크기는 다양하게 발생한다. 범프 결함의 경우 웨이퍼에 범프를 만드는 공정상의 문제로 발생하는 결함들로 poor reflow, undersized bump, missing bump 등의 결함들이 발생한다[4].

3. GPU

CPU의 다중코어(multi-core)와 GPU의 다수코어(many-core)형태의 프로세서가 보편화됨에 따라 병렬처리 프로그래밍의 중요성은 점점 더 커지고 있다. GPU상에서의 프로그래밍은 그래픽스 파이프라인 상에서 이루어져야 했기 때문에 프로그래머들의 진입장이 높은 편이었다. 이를 해결하기 위해 NVIDIA社에서 2007년 GPU 아키텍처인 CUDA를 발표하며, 파이프라인을 거치지 않고도 GPU를 활용할 수

그림 1. CPU와 GPU의 구조.

Fig. 1. Structure of CPU and GPU.

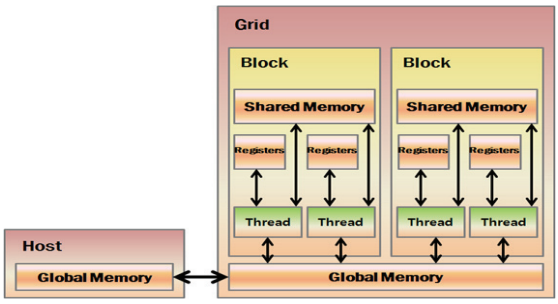


그림 2. GPU의 메모리 구조 및 CPU와의 데이터 교환.

Fig. 2. GPU memory structure and data exchange of CPU.

있는 프로그래밍 환경을 제공하게 되었다. 실제로 GPU의 구조는 작은 프로세서의 배열로 되어 있기 때문에 병렬 연산에서 탁월한 성능을 보여주고 있으며, 행렬 연산, 데이터 정렬(sort)등 기본 연산에서 이미 CPU보다 고속으로 처리가 가능하며, 다체문제(N-bidy problem),바이오 정보처리 등의 응용분야에서도 효과적임을 입증하는 다수의 논문이 발표되었다[5,6].

그림 1에 보여진 것처럼 CPU에서 한정적이었던 ALU 다수를 하나의 컨트롤러가 제어함으로써 부동소수점 연산, 산술 및 논리연산 능력에 관하여 CPU보다 성능이 우수하며, GPU와 CPU의 구조적인 차이로 인해 병렬처리가 가능하며, CPU에서 문제가 되었던 스레드간의 동기화 역시 CUDA에서 지원하는 간단한 함수로 손쉽게 동기화가 가능하다.

통상적으로 GPU 프로그래밍은 처리하고자 하는 데이터를 CPU에서 GPU의 전역메모리로 전송 후, GPU상에서 커널(kernel)이라 명명되는 함수로 처리하고 완료된 결과를 GPU의 전역메모리로부터 CPU로 가져오는 형태를 갖는다. 그림 2는 이러한 CPU와 GPU간의 데이터 교환 및 GPU의 메모리구조를 보다 쉽게 이해할 수 있도록 도와준다[7].

GPU는 다수의 CUDA 코어를 하나의 클러스터로 구성한 MP(Multiprocessors)로 구성된다. 원래 이 명칭은 SM (Streaming Processors)와 SP (Scalar Processor)로 불리었으나 CUDA 4.0버전이 발표되면서 SM을 MP로 SP를 CUDA코어로 명명하였다. 본 논문에서 사용된 NVIDIA社의 Tesla C2070 GPU는 32개의 CUDA 코어를 갖는 14개의 MP가 존재하며, 총 448개의 CUDA 코어를 병렬로 동작시킬 수 있으며, 총 6GB의 전역메모리를 갖고 있다. 여러 프로그램에서 실행되는 수백 개의 스레드를 효과적으로 관리하기 위해 CUDA 코어는 SIMT (Single-Instruction Multiple-thread) 아키텍처를 사용한다. CUDA 코어당 48개의 스레드를 하나로 묶는 워프(Warp)단위로 SIMD (Single-Instruction Multiple-Data)형태의 연산을 수행하며, 워프단위로 명령어가 동시에 실행된다[8].

III. 검사 알고리즘 고속화 및 병렬화

1. 검사 알고리즘

본 논문에서 제시하는 반도체 웨이퍼의 패턴검사 알고리즘은 실제 공정상에 사용이 가능하도록 만들기 위해 최대한 간단한 알고리즘으로 적용하였다. 또한 실제 적용시 알고리즘의 처리속도를 개선하기 위해 앞서 설명한 CUDA를 적용하여 병렬처리가 가능하도록 하여 처리 속도를 개선하였다.

웨이퍼 검사 알고리즘은 크게 두 단계로 나누어 볼 수 있는데, 첫 번째는 결함의 검사를 위한 영상개선 단계이며, 두 번째는 결함의 검사 단계이다. 영상을 취득 후 결함 검출을 위한 영상의 화질 개선이 이루어지게 된다. 이후 패턴의 주기 및 위치를 이용한 뺄셈연산을 수행하여, 영상간의 차이를 통해 결함을 검출한다. 이때 영상에서 발생할 수 있는 잡음을 제거하기 위하여 필터연산을 진행하여 결함의 오검출을 예방한다. 뺄셈연산 영상의 이진화(binarization)를 통하여 결함을 일차적으로 산출하고 차후 잡음을 제거한 필터링된 영상의 이진화한 영상간의 논리곱(AND)을 통해 결함에 대한 간단한 검증을 마친다.

그림 3은 위에서 설명한 검사 알고리즘의 전체 흐름도를 보인다.

보통 영상의 개선은 관심영역(region of interest)을 포함한 전체 영상의 화질 개선의 목적으로 사용되며, 이러한 영상개선 방법은 이미 많은 논문에 발표되고 사용되어져 왔다. 이러한 영상개선을 통해 결함과 비결함 영역에 대한 구분을 보다 명확하게 할 수 있을 뿐 아니라, 후의 단계인 영상패턴간의 뺄셈연산 단계에서 결함과 비결함 사이에 보다 높은 차이를 확보할 수 있는 중요한 단계이다.

그림 4는 본 논문에서 사용한 12K TDI 라인스캔 카메라를 통해 획득한 실제 결함 영상과 영상개선 효과를 보이고 있다. 그림 4의 결과에서 확인할 수 있듯이 개선된 영상과 개선되지 않은 영상간의 결함검출 과정을 시행했을 때 결함의 차이값은 크게는 4배 이상의 밝기차를 확인할 수 있

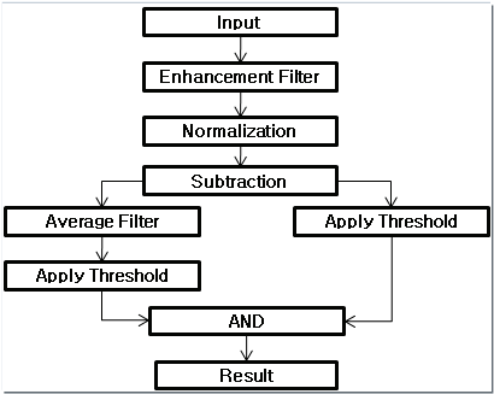


그림 3. 검사 알고리즘 순서도.

Fig. 3. Inspection algorithm flowchart.

그림 4. 원본 영상과 개선된 영상의 처리 결과.
Fig. 4. Processing result of original image and enhance image.

었다. 이런 영상 개선을 위해 필터연산을 사용하였는데 이러한 필터연산의 경우 영상에 존재하는 잡음도 같이 증폭되는 효과를 동반한다. 그렇기 때문에 개선이후 영상의 잡음을 없애주는 효과가 오검출된 결함의 일차적인 분류를 위한 후처리 연산으로서 평균필터를 사용하였다. 평균필터의 가장 큰 장점은 간단한 알고리즘 이외에도 알고리즘의 병렬 최적화 시에 필요한 연산의 간소화에 적합한 분할 연산이 가능하다는 것이다.

평균 필터를 통해 영상의 잡음을 제거한 후 실제 공정상에서 장비를 운용하는 조작자가 정할 수 있는 이진화의 임계값을 적용하여 각각의 결함을 검출한다. 이때 적용하는 이진화는 뺄셈연산 단계 이후 평균 필터 연산을 적용한 영상과 적용하지 않은 영상에 서로 다른 임계값을 적용하게 된다. 이때 임계값은 전적으로 실제 검사알고리즘이 적용되는 공정에 맞는 임계값을 사용한다.

그림 5는 그림 3에서 주어진 검사 알고리즘으로 검출한 결과 영상이며, 그림 5에서 볼 수 있듯이 개선된 원본영상의 검출 결함 크기가 실제 결함 크기와 비슷한 것을 확인할 수 있다. 이것은 영상자체에서 발생할 수 있는 문제, 혹은 검사 중간에 발생할 수 있는 불안정한 상황에서도 안정적인 결함 검출력을 확보할 수 있음을 알 수 있다.

그림 5. 검출된 결함 영상.
Fig. 5. Detected defect images.

표 1. C와 CUDA 알고리즘 설계.
Table 1. C and CUDA algorithm design.

C 알고리즘 설계	CUDA 알고리즘 설계
Enhancement Filter	Enhancement Filter
Normalization	
뺄셈연산	뺄셈연산
뺄셈연산 영상 이진화	
평균필터	평균필터
평균필터 영상 이진화	
AND	AND

2. 알고리즘의 CUDA 설계

앞서 설명한 CUDA를 이용하여 반도체 웨이퍼 패턴검사 알고리즘을 CUDA에 적용하기 위해 CUDA에서 활성화가 가능한 최대 스레드를 활성화 시키고, CUDA 코어를 전부 활용 가능한 최대 Grid와 Block을 기준으로 적용하였다. 그에 따라 최대 스레드의 활성화를 기준으로 설계하였고, 실험을 통해 이를 검증하였다. 프로그램의 구성은 총 4개의 커널로 분할하여 설계하였으며, 동일한 Grid와 Block에서 동작해야 한다면 영상의 크기 및 하드웨어의 변경이 있을 때 프로그램을 항상 수정해주어야 한다는 번거로움이 존재하지만 본 논문은 최대한 자동화에 초점을 맞추어 프로그램 설계 및 구현하였다.

표 1은 C 알고리즘과 CUDA 알고리즘의 설계에 대한 내용으로, C알고리즘에서의 영상개선과 정규화 과정을 CUDA 알고리즘에서 영상개선으로 통합하였고, 뺄셈연산과 뺄셈연산후 이진화 과정을 뺄셈연산과정으로 통합, 평균필터 연산과 평균필터 연산후 이진화 과정을 평균필터 과정으로 통합하였으며, 마지막 논리곱 과정을 동일하게 적용하였다.

IV. 병렬 알고리즘의 최적화

C로 구현한 알고리즘을 CUDA로 적용 완료 후 일차적으로 성능을 검증해야 한다. 단순히 C코드를 CUDA로 변환한 것만으로 병렬연산에 적합한 GPU에서의 성능향상을 기대할 수 없다. 그렇기 때문에 알고리즘에서 가장 최대로 지연되는 구간을 선정하고 선정된 구간의 최적화로 알고리즘 전체적인 성능을 향상시킬 수 있다. 그러기 위해서 CUDA로 구현된 각 단계에서의 성능을 측정하고 Grid와 Block을 나누어 각각의 커널 수행시간을 측정하여 지연구간을 설정하였다.

표 2는 GPU에서 Grid와 Block의 변화에 따라 구현한 CUDA 알고리즘의 커널당 수행시간이다. 사용된 각각 Gird와 Block은 1차원 형태인 Grid(448,1)와 Block(1024,1)의 설계부터 448과 1024가 넘지 않는 범위 내에서 Grid와 Block을 변경하였다. 실험결과 각 커널의 수행시간이 Grid(7,64) / Block(32,32)과 Grid(7,64)/Block(64,16)에서 최소 수행시간이 측정되는 것을 확인할 수 있었으며, 커널별 최대 30ms이상 차이를 보이는 것을 확인할 수 있었다. 이 데이터를 통해 필터연산과 평균필터 연산구간을 병렬 최적화 구간으로 선정하였다.

표 2. CUDA를 적용한 검사 알고리즘 성능 및 지연구간.

Table 2. Performance of CUDA Inspection algorithm and delay section.

448		1024		12000 x 52571					
Grid_X	Grid_Y	Block_X	Block_Y	Enhance	Sub	Average	Th 1	AND	Total
448	1	1024	1	665.15	304.94	347.08	304.64	300.46	1922.25
224	2	1024	1	363.19	157.95	193.88	157.32	155.09	1027.43
112	4	1024	1	232.21	90.48	134.19	91.01	80.56	628.45
56	8	1024	1	147.81	50.56	91.57	50.99	43.66	384.60
28	16	1024	1	106.52	31.43	69.58	31.46	24.34	263.32
14	32	1024	1	85.29	21.60	58.54	21.68	14.78	201.88
7	64	1024	1	81.27	18.90	56.13	18.74	12.10	187.13
448	1	512	2	364.04	158.07	194.70	157.40	153.69	1027.91
224	2	512	2	215.94	86.05	120.65	85.86	81.47	589.97
112	4	512	2	152.48	50.62	93.81	50.61	43.92	391.43
56	8	512	2	108.90	31.29	70.87	31.11	24.58	266.75
28	16	512	2	87.41	21.74	59.20	21.39	14.97	204.71
14	32	512	2	82.40	18.94	56.77	18.47	12.21	188.79
7	64	512	2	77.05	17.31	53.84	17.05	10.31	175.56
448	1	64	16	102.13	31.02	61.26	31.90	25.19	251.50
224	2	64	16	83.20	23.21	51.74	23.23	16.10	197.47
112	4	64	16	80.86	20.02	50.67	19.72	12.78	184.05
56	8	64	16	75.35	19.28	48.34	19.15	11.58	173.69
28	16	64	16	72.67	19.08	46.79	18.01	11.07	167.62
14	32	64	16	71.12	18.65	46.97	17.83	10.52	165.09
7	64	64	16	70.41	14.41	44.37	13.63	7.96	150.79
448	1	32	32	82.93	29.27	50.82	26.27	19.91	209.20
224	2	32	32	79.58	29.30	48.97	24.00	17.22	199.07
112	4	32	32	75.05	28.46	47.24	23.23	16.84	190.82
56	8	32	32	72.28	27.07	46.25	22.33	15.84	183.76
28	16	32	32	70.62	27.50	44.80	21.92	15.73	180.58
14	32	32	32	71.01	16.43	45.71	15.09	9.40	157.64
7	64	32	32	70.83	14.94	44.17	14.02	8.60	152.55
448	1	16	64	83.38	53.98	57.14	35.89	28.89	259.28
224	2	16	64	82.22	57.25	60.67	39.69	32.51	272.34
112	4	16	64	80.35	56.83	62.23	40.44	31.54	271.39
56	8	16	64	79.69	56.63	60.52	38.24	31.01	266.10
28	16	16	64	77.87	53.50	59.92	21.12	26.88	239.30
14	32	16	64	77.27	52.15	60.77	22.60	24.10	236.89
7	64	16	64	70.47	42.24	50.20	18.05	16.91	197.87
Min				70.41	14.41	44.17	13.63	7.96	150.787

1. 지연구간 병렬 최적화

CUDA를 이용한 병렬 프로그래밍의 최적화 방법은 이미 많은 연구들이 진행되고 있다[9]. 가장 일반적으로 CUDA 성능의 향상을 알기 위해서는 활성 워프(active warp)의 활용과 스레드 사용률의 극대화 등이 있으며, CUDA에서 사용되는 공유메모리의 사용을 시작으로 공유메모리의 뱅크 충돌을 방지하는 방법, 메모리를 로딩할 때 선행계산을 함으로써 지연시간을 숨기는 기술, 전역메모리의 엑세스를 결합하여 메모리 대역폭(memory bandwidth)을 최대한 이용할 수 있게 하는 기술, 메모리, 명령어의 overhead로 인한 병목지점의 파악, if 조건문으로 발생하는 스레드 분기의 발산을 막는 방법, 조건문과 loop문의 코드를 풀어 작성하는 loop unroll 기법 등이 있다. 하지만 위의 기술들 외에 알고리즘 자체의 최적화를 통해 수행시간의 감소를 확인할 수 있으며, 본 논문에서는 알고리즘의 최적화 및 연산의 간소화와 더불어 loop unroll 기법을 적용하여 최적화를 진행하였다.

앞서 지연구간 설정에서 영상 개선 필터와 평균필터의 일반적인 방법은 for문을 활용한 연산이나 마스크가 separable하다면 N x N번의 연산을 하지 않고 N + N번의 연산으로 동일한 결과를 도출할 수 있으며 이런 separable filter의 고속화에 관한 연구도 활발히 진행되고 있다[10].

일반적으로 image convolution은 마스크의 곱과 결과값의 총 합으로 진행한다. 하지만 그림 6에서 확인할 수 있듯이 마스크의 크기가 커질수록 연산 횟수가 많아지면서 그와 비례하여 처리시간도 증가하게 된다. 또한 이런 image convolution이 안고 있는 문제점들로 인해 실제 검사장비나 공정상에서의 사용이 제약되는 문제를 야기하고 있다. 연산의 간소화를 통해 처리시간의 단축을 가져올 수 있는 방법

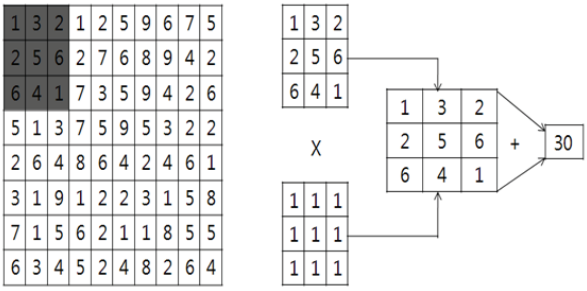


그림 6. 일반적인 image convolution.

Fig. 6. General image convolution.

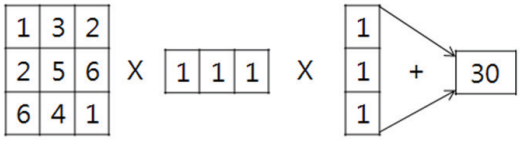


그림 7. Separable image convolution의 개요.

Fig. 7. Outline of separable image convolution.

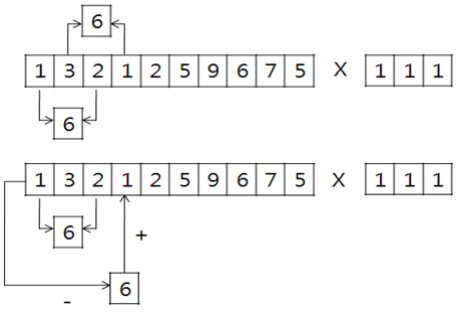


그림 8. 일반적인 연산과 최적화에 사용된 연산.

Fig. 8. General algorithm and using algorithm for optimization.

이 separable filter의 고속 연산이다.

이미 많은 필터연산에 적용되어 있는 이 방법은 필터연산의 고속화에 적용되었고, 본 논문에서 사용하는 필터에 적용하였다[11]. 그림 7은 평균필터를 2개의 1차원 형태의 마스크로 분리하여 적용한 것을 나타내었다. 그림 7에서 확인 가능하듯 통상적인 평균필터는 9번의 덧셈 그리고 1번의 나눗셈을 진행하는 것이 일반적이나, separable 특징을 활용하여 1차원 형태로 분리된 경우 각각 3번의 덧셈을 진행, 총 6번의 덧셈을 통해서 동일한 결과를 산출할 수 있다. 마스크의 크기가 커질수록, 대상 영상이 클수록 그 효과는 커진다 할 수 있다.

다음으로 필터연산의 간소화를 진행하였다. 그림 8은 연산 횟수를 줄이기 위해 적용된 계산법을 나타낸 것이다. 일반적으로 마스크의 크기의 횟수만큼 반복된 덧셈을 통해 결과를 얻는다. 하지만 본 논문에 적용된 계산법은 처음 3개의 값만 3번의 덧셈을 진행하고 그 이후는 한번의 뺄셈과 다음차례 값의 덧셈을 통해 연산을 간소화 하였다. 이것은 3 x 3 크기 마스크의 경우 가시적으로 1번의 연산 감소만을 가져 오지만 마스크의 크기가 커질수록 연산의 간소화 수치는 커지며, 적용영상의 크기가 커질수록 그 감소의

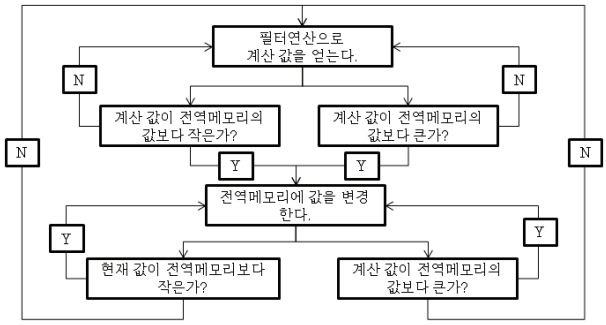


그림 9. 정규화를 위한 데이터 종합 알고리즘.

Fig. 9. Data synthesis algorithm for normalization.

차이는 더 커진다 할 수 있다. 연산을 진행 후 정규화 과정이 필요하다. 영상 전체의 최솟값과 최댓값을 이용한 정규화 과정은 CUDA를 이용한 병렬처리시 각 스레드간 데이터 교환이 불가능하기 때문에 필터연산을 진행 후 영역별 최대 및 최솟값을 구하고, 이후 스레드별 최솟값과 최댓값을 종합하는 과정이 필요하다. 즉, 총 458,752개의 스레드를 생성하는 본 논문에서는 각 스레드별로 최댓값과 최솟값의 데이터를 찾아내 917,504개의 데이터 중 최댓값과 최솟값을 찾아내는 과정이 필요하다. 이러한 과정을 최소한으로 줄이기 위해서 필터 연산값을 전역변수와 비교해 업데이트 하는 형식을 취한다. 하지만 458,752개의 스레드가 동시에 접속하여 데이터를 업데이트할 때에 최솟값 혹은 최댓값을 비교할 때 다른 스레드가 접속하여 값을 변경할 수 있는 경우를 방지하기 위하여 변경된 값과 스레드의 값을 비교하여 최소 혹은 최댓값을 확인할 때까지 변경을 하게 된다. 이후 중복계산을 방지하기 위하여 __syncthread() 함수를 통하여 모든 스레드의 동기화를 통해 데이터의 종합이 완료 후 연산을 진행하게 된다.

그림 9는 정규화를 진행하기 위한 순서도를 나타낸다. 이 방법을 통해 각 스레드에서 얻어진 917,504개 데이터를 종합하는 과정이 생략된다.

마지막으로 사용된 방법은 loop unroll기법이다[12]. Loop unroll의 목적은 loop overhead를 줄이며, 캐시 히트율의 상승을 도모하는 것이 주목적이다. 그림 10은 sum = sum + x[i] + y[i]의 작업을 수행하는 일반적인 코드이다. 하지만 이것을 loop unroll기법을 사용하여 코드를 재 작성하면, 그림 11과 같이 나타낼 수 있다. 가시적으로는 loop unroll 기법을 적용한 코드가 그렇지 않은 코드보다 길어진 것을 확인할 수 있다. 하지만 앞의 두가지 코드순서를 비교하게 되면 표 3과 같이 표현할 수 있다.

표 3을 기준으로 100회의 동일한 작업 횟수를 비교한다면, 일반코드에서 ①번과 ②번은 균등하게 100번씩 작업을 하게 된다. 하지만 loop unroll기법이 적용된 코드는 ①번 작업은 ②번부터 ⑨번의 작업을 수행 후에 하게 되므로, ①번 작업은 약 12번(①번 + ②번 작업에 모두 9번을 소비하므로 100/9 +1), ②번 작업은 약 88번 수행하게 된다. 특히 ①번 작업 같이 if를 사용하는 작업은 클럭의 소모율이 심하기 때문에 loop unroll 기법을 사용하면, 전반적인 클럭 효율이 상승하게 된다.

```
void normal_add(u_int *x, u_int* y,u_int size, u_int *result)
{
        u_int sum=0;
        u_int i=0;
        for(i=0;i<size;i++)
        {
                sum += x[i] + y[i];
        }
        *result = sum;
}
```

그림 10. sum = sum + x[i] + y[i]의 일반적인 코드.

Fig. 10. General code of sum = sum + x[i] + y[i].

```
void unroll_add(u_int *x, u_int* y, u_int size, u_int *result)
{
        u_int sum=0;
        u_int i=0;
        u_int ssize = (size / 8) * 8; //8번씩 할 것이므로 8의 배수로 만듬

        for(i=0;i<ssize;i+=8) //한번 루프에 8번씩 반복한다.
        {
                sum += x[i  ] + y[i  ];
                sum += x[i+1] + y[i+1];
                sum += x[i+2] + y[i+2];
                sum += x[i+3] + y[i+3];

                sum += x[i+4] + y[i+4];
                sum += x[i+5] + y[i+5];
                sum += x[i+6] + y[i+6];
                sum += x[i+7] + y[i+7];

        }
    for(i; i < size; i++) //나머지 부분
    {
            sum += x[i] + y[i];
    }
    *result = sum;
}
```

그림 11. Loop unroll기법을 적용한 코드.

Fig. 11. Applied code of loop unroll technique.

표 3. 일반적인 코드와 loop unroll 코드의 실행순서.

Table 3. Flowchart of General code and loop unroll code.

코드 수행 / 순서	일반적인 코드	loop unroll 코드
①	for에서 i < size가 만족하면 i가 1 증가	for에서 i < size가 만족하면 i가 8 증가
②	sum += x[i] + y[i]	sum += x[i] + y[i]
③	①번으로 돌아간다.	sum += x[i+1] + y[i+1]
④		sum += x[i+2] + y[i+2]
⑤		sum += x[i+3] + y[i+3]
⑥		sum += x[i+4] + y[i+4]
⑦		sum += x[i+5] + y[i+5]
⑧		sum += x[i+6] + y[i+6]
⑨		sum += x[i+7] + y[i+7]
⑩		①번으로 돌아간다.

V. 실험 및 결과

제안한 방법에 대한 검증을 위한 실험은 제안된 알고리즘을 GPU를 통해 구현하고, 알고리즘의 정확한 구현이 되

었는지에 관한 CPU와의 비교실험을 통해 검증하고, 결함의 유무를 판정할수 있는 임계값 선정 방법에 대한 실험을 진행하고, 마스크 크기에 따른 검출 성능 실험을 통해 최적의 마스크 크기를 선정하는 실험, 구현된 CUDA 알고리즘의 커널별 처리속도의 지연구간 선정을 통해 결정되어진 구간의 최적화를 통한 GPU의 처리속도 향상에 대한 실험, 마지막으로 CPU와 GPU와의 성능 비교를 통한 실험으로 나누어 진행하였다.

실험에 사용된 CPU는 Intel(R) Core(TM)2 Quad CPU Q9550 @2.83GHz와 2GB의 RAM을 갖는 CPU이고, GPU는 Tesla C2070으로 448개의 CUDA코어와 6GB의 메모리를 갖는 GPU를 사용하였다. 실험영상은 TDI 12K line scan camera로 획득한 실제 반도체 웨이퍼 패턴 영상을 사용하였으며, 영상의 크기는 12,000 x 52,571픽셀이고 총 3가지의 패턴을 갖는 영상을 실험에 사용하였다.

1. GPU 알고리즘 구현 검증 실험

실험의 첫 번째로 GPU로 구현한 알고리즘의 검증을 하였다. 알고리즘의 순서별로 영상에 적용한 뒤 CPU와 결과를 비교하고 그 차이가 0이 된다면 CPU와 GPU의 알고리즘은 동일하다고 할 수 있다. 하지만 차이값이 0이 아닌 경우 GPU를 이용한 알고리즘의 구현이 문제가 있다는 것을 알 수 있다.

가장 먼저 알고리즘의 가장 첫 단계인 영상개선 부분이다. 전체영상을 개선한 결과를 기준으로 CPU와 GPU의 차

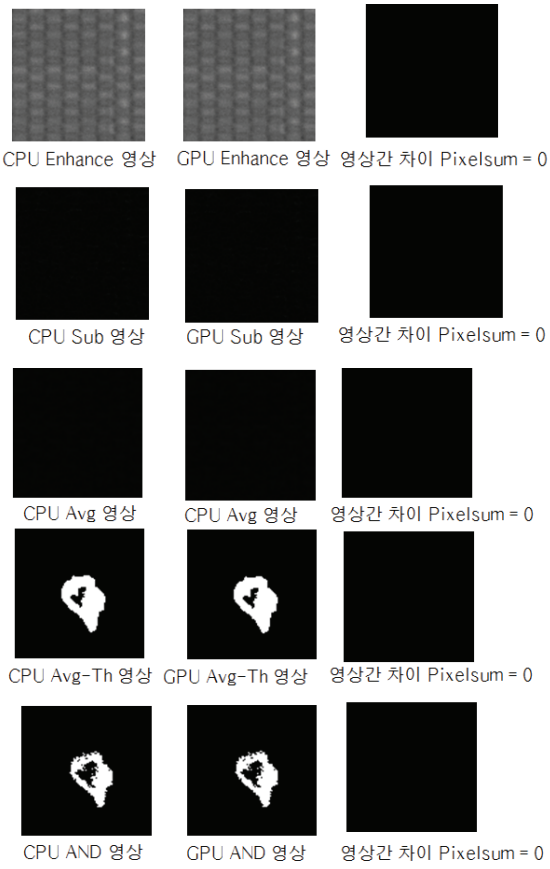


그림 12. GPU와 CPU의 구현 검증 실험 결과.

Fig. 12. Experimental results verified the GPU and CPU.

이값을 얻은 두 영상간의 뺄셈연산과 결과 영상 전체에 대한 합을 통해서 검증하였다. 위와 동일한 실험 방법으로 알고리즘의 각 순서별로 실험한 결과, 그림 12와 같이 CPU와 GPU에서 구현된 알고리즘은 동일한 결과를 도출해 낸다는 것을 알 수 있었다.

2. GPU의 결함 검출 실험

GPU의 결함 검출 성능을 실험하기 위해 우선 알고리즘에 적합한 임계값 선정실험을 실시하였다. 알고리즘에서 변경이 가능한 파라미터는 총 3가지가 존재한다. 평균필터의 마스크의 크기와 이진화의 임계값, 그리고 뺄셈연산 후 이진화의 임계값, 총 3가지의 파라미터값의 변경이 가능하다. 이들 3개의 파라미터를 대상으로 실험 영상에 적용하여 결함의 검출 성능에 대한 실험을 진행하였다. 마스크의 크기는 검출하고자하는 결함의 크기와 영상의 화질에 따라 다른 값을 사용해야 하므로, 현재 사용하는 영상에 적합한 마스크의 크기를 결정하기 위해 3×3, 5×5, 7×7로 변화시키면서 임계값의 변화에 따라서 결함의 미검률과 과검률을 측정하고 분석하여 적합한 마스크의 크기와 각 단계의 이진화 임계값을 결정하였다.

실험결과를 분석한 결과 7×7 마스크 보다 3×3 마스크와 5×5 마스크의 결함의 과검률과 미검률이 상대적으로 낮았다. 7×7마스크의 경우 전체적인 실험에서 항상 10% 이상의 높은 미검률과 과검률을 유지하고 있는 반면, 3×3 마스크와 5×5 마스크의 경우 평균필터 이진화 임계값이 33~35에서 미검률은 0%이나, 과검률은 각각 6%, 2%, 1%로 7×7 마스크보다 상대적으로 안정적인 수치를 유지하고 있으며, 임계값 36과 37에서 과검률과 미검률이 0%를 기록하였다. 하지만 평균필터 이진화 임계값 36의 실험 결과를 볼 때 5×5 마스크 보다 3×3 마스크의 과검률과 미검률이 좀 더 안정적인 수치를 나태고 있음을 확인할 수 있다. 위 실험 결과를 토대로 적합한 마스크의 크기는 3×3 각 필터의 이진화 임계값은 36임을 확인할 수 있었다.

3. 지연구간 최적화를 통한 처리속도 향상실험

지연구간 최적화를 통한 속도향상실험을 위해 앞서 설명한 알고리즘의 최적화 및 연산의 간소화, loop unroll 기법을 적용하여 최적화를 진행하였다. 그리고 앞서 실함한 알고리즘의 최적 파라미터를 적용하여 단계별 최적화를 진행하였다. 표 4와 표 5는 최적 파라미터 선정 후 GPU 보드상에서 Block의 수와 스레드의 수를 변화시키면서 얻은 최적화 전후의 실험 결과이다.

표 4와 표 5에서 확인할 수 있듯이 알고리즘의 최적화

표 4. 3×3마스크의 최적화전 수행시간.

Table 4. Processing time of 3×3 mask before optimization.

(단위: ms)

448		1024		12000 x 52571						
Grid_X	Grid_Y	Block_X	Block_Y	영상개선	Sub	평균필터	Sub Th	평균 Th	And	Total
7	64	64	16	70.41	14.41	44.37	6.82	7.82	7.96	151.79
448	1	32	32	82.93	29.27	50.82	13.14	14.14	19.91	210.20
224	2	32	32	79.58	29.30	48.97	12.00	13.00	17.22	200.07
112	4	32	32	75.05	28.46	47.24	11.61	12.61	16.84	191.82
56	8	32	32	72.28	27.07	46.25	11.16	12.16	15.84	184.76
28	16	32	32	70.62	27.50	44.80	10.96	11.96	15.73	181.58
14	32	32	32	71.01	16.43	45.71	7.55	8.55	9.40	158.64
7	64	32	32	70.83	14.94	44.17	7.01	8.01	8.60	153.55
Min				70.41	14.41	44.17	6.82	7.82	7.96	151.787

표 5. 3×3마스크의 최적화후 수행시간.

Table 5. Processing time of 3×3 mask after optimization.

(단위: ms)

448		1024		12,000 x 52,571						
Block_X	Block_Y	Thread_X	Thread_Y	영상개선	Sub	평균필터	Sub Th	평균 Th	And	Total
7	64	64	16	23.99	7.20	33.49	3.40	3.70	3.97	75.76
448	1	32	32	30.89	14.80	35.27	6.52	6.82	10.05	104.34
224	2	32	32	29.72	14.33	35.37	5.77	6.07	8.38	99.65
112	4	32	32	27.18	14.22	37.33	5.68	5.98	8.10	98.50
56	8	32	32	26.36	13.65	36.54	5.45	5.75	7.94	95.69
28	16	32	32	25.69	13.92	35.55	5.45	5.75	8.26	94.64
14	32	32	32	25.03	8.21	35.60	3.77	4.07	4.69	81.38
7	64	32	32	23.96	7.47	33.02	3.51	3.81	4.29	76.05
Min				23.96	7.20	33.02	3.40	3.70	3.97	75.76

표 6. CPU와 GPU의 성능비교.

Table 6. Comparison CPU with GPU.

단계 / 구분	Enhance	Sub	Average	Th1	AND	Total
CPU	13,520ms	1,145ms	658ms	3,258ms	2,631ms	21,212ms
GPU	23.99ms	7.20ms	33.49ms	7.5ms	3.97ms	75.76ms

및 연산의 간소화, loop unroll 기법을 적용하여 최적화를 하였을 때 성능향상을 확인할 수 있었다. 특히 약 70ms가 소요되었던 영상개선필터 과정에서 23ms로 향상되었으며, 평균필터의 경우 44ms에서 33ms로 향상되었다. 전체적인 수행시간은 최적화전 151.79ms의 수행시간이 최적화후 75.76ms로 약 50% 이상의 성능향상을 확인할 수 있었다.

4. CPU와 GPU의 성능비교 실험

CPU와 GPU의 성능 비교실험을 하기 위해서 앞의 실험에서 알 수 있던 최적의의 임계갑과 마스크의 크기를 CPU와 GPU에 동일하게 적용하고 실제로 두 성능을 비교 하였다.

표 6에서 알 수 있듯이, CPU와 GPU의 성능의 차이는 21,212ms에 비해 약 210배 이상 성능이 향상된 75ms의 처리 속도 향상을 확인할 수 있었다.

VI. 결론

미세화 되는 웨이퍼 패턴은 최대 30um까지 검사가 가능한 광학계를 요구하는 실정이며, 그에 따라 영상의 크기와 데이터의 크기가 대용량화 되어 실시간으로 웨이퍼를 검사하는 알고리즘의 부재가 나타나고 있다. 이에 따라 본 논문은 GPU를 이용한 반도체 웨이퍼 패턴검사 알고리즘의 고속화에 대해 제안하였다. 실제 산업에서 사용될 수 있는 알고리즘과 그에 따른 대용량 데이터처리에 맞춰 GPU를 이용하였고, 성능의 향상을 위한 알고리즘의 개별 최적화 및 병렬화를 진행하였다. 연산의 간소화와 loop unroll 기법을 이용해 지연구간 최적화를 통해 영상 개선필터에서 70ms 소요 되었던 시간을 개선후 23.99ms로 약 50% 이상 성능을 향상시켰으며, 뺄셈연산에 소요되었던 14.41ms를 7.20ms로 50% 이상 성능을 향상시켰으며, 전체적으로 151.79ms의 수행시간이 최적화후 75.76ms로 약 50% 이상의 고속화됨을 확인할 수 있었다. 특히 CPU와의 비교 실험에서는 전체 21,212ms에 비해 약 210배 이상 수행시간이 향상된 75ms의 처리 속도 향상을 확인할 수 있었다.

REFERENCES

[1] J. S. Lee, “Automatic classification algorithm of defects in semiconductor package molding surface inspection using pattern recognition,” *M. S. Thesis (in Korean)*, Hoseo University, 2009.

[2] R. Bertz and P. Leahy, “Inspection Challenges of Leadless Packages,” *Proc. SEMOCPN*, pp. 418-422, 2002.

[3] S. J. Nam and K. S. Hahn, “Implementation of automated defect detection and classification system for semiconductor wafers,” *Proc. of the 28th KISS Fall Comference (in Korean)*, vol. 28, no. 2, pp. 334-336, 2001.

[4] K. S. Jang and C. H. Jeon “Classification rule-based defect pattern detection of semiconductor wafer map,” *Proc. of 2005 KIIE Fall Conference (in Korean)*, pp. 131-139, 2005.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, “A performance study of general-purpose applications on graphics processors using CUDA,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370-1380, 2008.

[6] Y. G. Yeom and Y. K. Cho, “High-speed implementation of block ciphers on graphics processing units using CUDA library,” *Journal of the Korea Institute of Information Security and Cryptology (in Korean)*, vol. 18, no. 3, pp. 23-31, 2008.

[7] M. Moazeni, A. Bui, and M. Sarrafzadeh, “A memory optimization technique for software-managed scratchpad memory if GPUs,” *2009 IEEE 7th Symposium on Application Specific Porcessors*, pp. 43-49, 2009.

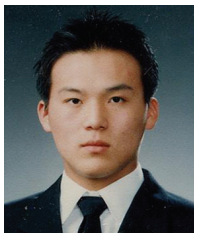
[8] NVIDIA, *C. U. D. A. NVIDIA CUDA Programming Guide.* 2011.

[9] T. J. Park, J. M. Woo, and C. H. Kim, “CUDA-based parallel bi-conjugate gradient matrix solver for BioFET simulation,” *Journal of the Institute of Electronics Engineers of Korea (in Korean)*, vol. 48, no. 1, pp. 90-100, 2011.

[10] G. Vialaneix and T. Boubekeur, “SBL mesh filter: fast separable approximation of bilateral mesh filtering,” *ACM SIGGRAPH 2011 Talks. ACM*, p. 24, 2011.

[11] V. Areekul, U. Watchareeruetai, and S. Tantaratana, “Fast separable gabor filter for fingerprint enhancement,” *Biometric Authentication. Springer Berlin Heidelberg*, pp. 403-409, 2004.

[12] S. Kurra, N. K. Singh, and P. R. Panda, “The impact of loop unrolling on controller delay in high level synthesis,” *In IEEE Design, Automation & Test in Europe Conference & Exhibition*, pp. 1-6, 2007.



박 영 대

2009년 호서대학교 전자공학과 졸업. 2012년 동 대학원 석사. 2012년 2월~2012년 12월 ㈜세미시스코 주임. 2012년 12월~2013년 9월 ㈜쓰리디플러스 대리. 2013년 9월~현재 ㈜에이앤아이 기술연구소 주임연구원. 관심분야는 영상처리, 머신비전, 병렬처리 등.

김 준 식

1987년 서강대학교 전자공학과 졸업. 1989년 동 대학원 석사. 1993년 동 대학원 박사. 1993년~1994년 서강대학교 부설산업기술연구소 박사후연구원. 2007년~2008년 Southern Oregon Univ. 방문교수. 1994년~현재 호서대학교 전자공학과 교수. 관심분야는 영상신호처리, 컴퓨터 비전, 패턴인식, 영상압축 및 통신, 반도체/디스플레이 검사장비, 병렬처리 등.

주 효 남

1976년 서울대학교 전기공학과 졸업. 1976년~1982년 국방과학연구소 선임연구원. 1985년 Virginia Polytechnic Institute & State Univ. VA, USA 석사. 1985년~1987년 Machine Vision International 선임연구원. 1990년~1996년 The Boeing Company Principal Engineer. 1991년 Univ. of Washington 전기전자공학 박사. 1996년~2000년 삼성전자 생산기술센터 자동화연구소 연구소장. 2000년~2002년 ㈜넥스트아이 연구개발부문 사장. 2002년~현재 호서대학교 디스플레이공학부 교수. 관심분야는 영상신호처리, 컴퓨터 비전, 패턴인식, 반도체/디스플레이 검사장비, 병렬처리 등.