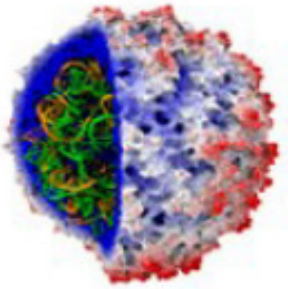
31st ACM/SIGAPP Symposium on Applied Computing

SMT-Based Context-Bounded Model Checking for CUDA Programs

Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Vanessa Santos, Celso Barbosa, Ricardo Ferreira, and **Lucas Cordeiro**

CUDA: parallel computing platform and API model

- Developed by NVIDIA to configure GPUs

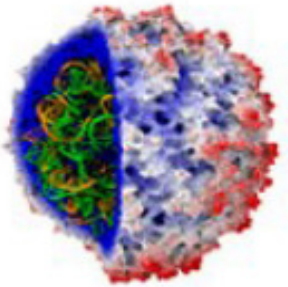


CUDA: parallel computing platform and API model

- Developed by NVIDIA to configure GPUs

  – initially used in **graphical processing** in **games** applications

    ➢ specially those that require **high computational power**



CUDA: parallel computing platform and API model

- Developed by NVIDIA to configure GPUs

  – initially used in **graphical processing** in **games** applications

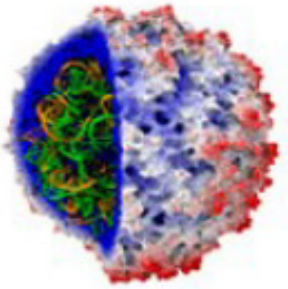
    ➢ specially those that require **high computational power**

  – Currently used in:

    ➢ biomedicine

    ➢ air traffic control

    ➢ weather simulation



CUDA: parallel computing platform and API model

- Developed by NVIDIA to configure GPUs

  – initially used in **graphical processing** in **games** applications

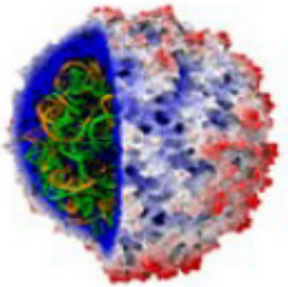
    ➢ specially those that require **high computational power**

  – Currently used in:

    ➢ biomedicine

    ➢ air traffic control

    ➢ weather simulation



- We need to ensure **code correctness** in safety-critical GPU applications

Typical Programming Errors in CUDA

- CUDA-based C/C++ programs are subject to
  - Arithmetic under- and overflow, buffer overflow, pointer safety, and division by zero

Typical Programming Errors in CUDA

- CUDA-based C/C++ programs are subject to
  - Arithmetic under- and overflow, buffer overflow, pointer safety, and division by zero
  - Data race conditions, shared memory, and barrier divergence

Typical Programming Errors in CUDA

- CUDA-based C/C++ programs are subject to
  - Arithmetic under- and overflow, buffer overflow, pointer safety, and division by zero
  - Data race conditions, shared memory, and barrier divergence
    - lead to incorrect results during the program execution
    - they are hard to detect due to the parallel operations

Typical Programming Errors in CUDA

- CUDA-based C/C++ programs are subject to
  - Arithmetic under- and overflow, buffer overflow, pointer safety, and division by zero
  - Data race conditions, shared memory, and barrier divergence
    - ➢ lead to incorrect results during the program execution
    - ➢ they are hard to detect due to the parallel operations

Array out-of-bounds due to incorrect access in unallocated memory region

```
int a[2];
...
kernel(int *a){
    if(a[1]==1)
        a[threadIdx.x+2] = threadIdx.x ;
    else
        a[threadIdx.x] = threadIdx.x;
}
```

Objectives of this work

Exploit SMT-based context-BMC to verify CUDA-based programs

Objectives of this work

Exploit SMT-based context-BMC to verify CUDA-based programs

- Develop an **operational model** for the CUDA platform (named COM)
  - Integrate COM into the *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) (TSE'12)

Objectives of this work

Exploit SMT-based context-BMC to verify CUDA-based programs

- Develop an **operational model** for the CUDA platform (named COM)
  - Integrate COM into the *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) (TSE'12)
- Apply **context-bounded model checking** based on the Satisfiability Modulo Theories (SMT)
  - Monotonic Partial Order Reduction (MPOR) (CAV'09)

Objectives of this work

Exploit SMT-based context-BMC to verify CUDA-based programs

- Develop an **operational model** for the CUDA platform (named COM)
  - Integrate COM into the *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) (TSE'12)
- Apply **context-bounded model checking** based on the Satisfiability Modulo Theories (SMT)
  - Monotonic Partial Order Reduction (MPOR) (CAV'09)
- Compare **ESBMC-GPU** experimental results with other **state-of-art software verifiers** for CUDA

CUDA Operational Model (COM)

- COM aims to
  - **Abstractly** represent the associated CUDA libraries
    - checks **pre-** and **post-conditions**
    - simulates **behavior**
  - Reduce verification effort
    - by only checking **relevant behavior**

CUDA Operational Model (COM)

- COM aims to
  - **Abstractly** represent the associated CUDA libraries
    - checks **pre-** and **post-conditions**
    - simulates **behavior**
  - Reduce verification effort
    - by only checking **relevant behavior**
- COM allows ESBMC to check **specific properties** related to CUDA libraries

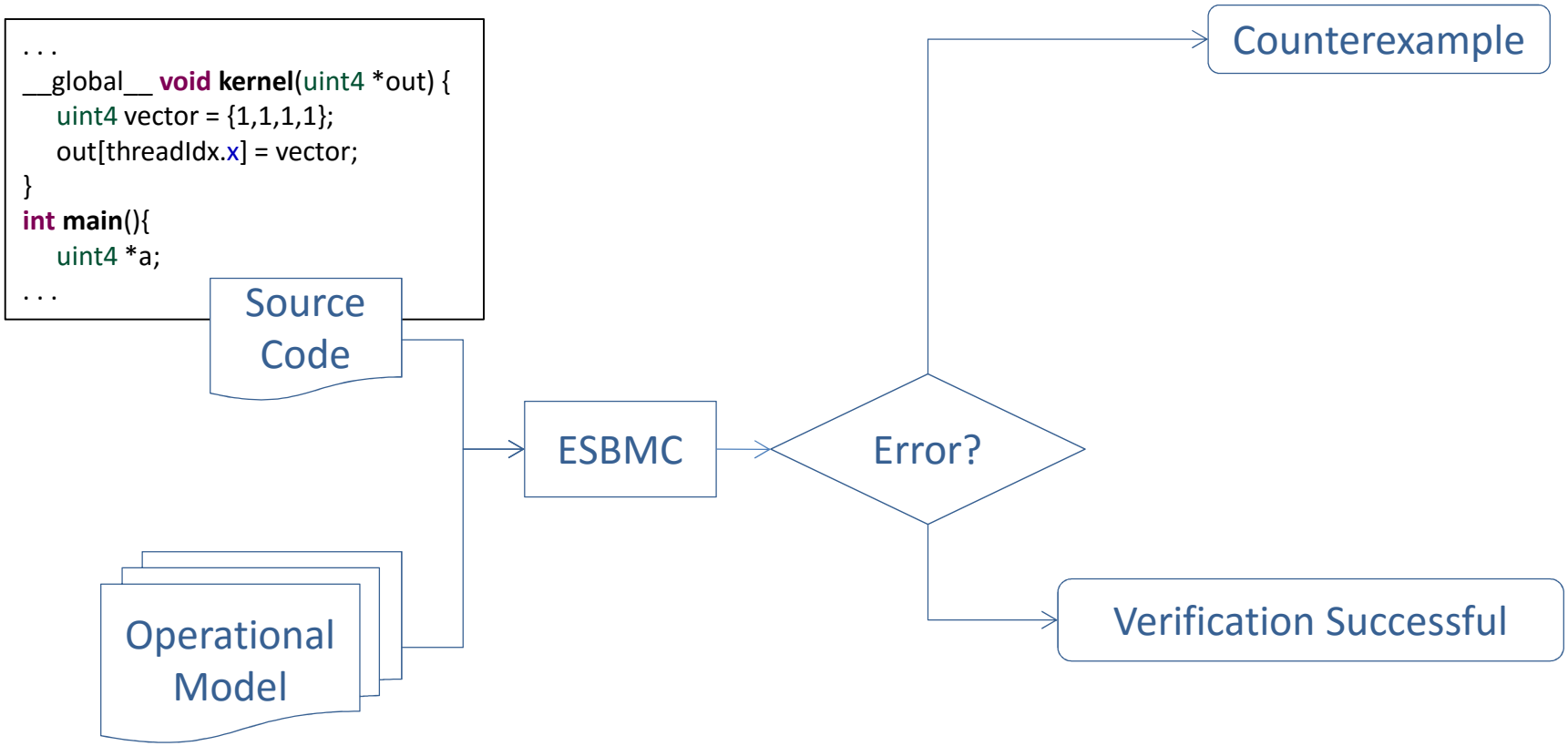
CUDA Operational Model (COM)

- COM aims to
  - **Abstractly** represent the associated CUDA libraries
    - checks **pre-** and **post-conditions**
    - simulates **behavior**
  - Reduce verification effort
    - by only checking **relevant behavior**
- COM allows ESBMC to check **specific properties** related to CUDA libraries
  - Other extensions to ESBMC based on operational models
    - ESBMC++ (ECBS'13) and ESBMC^{QtOM} (SPIN'16)

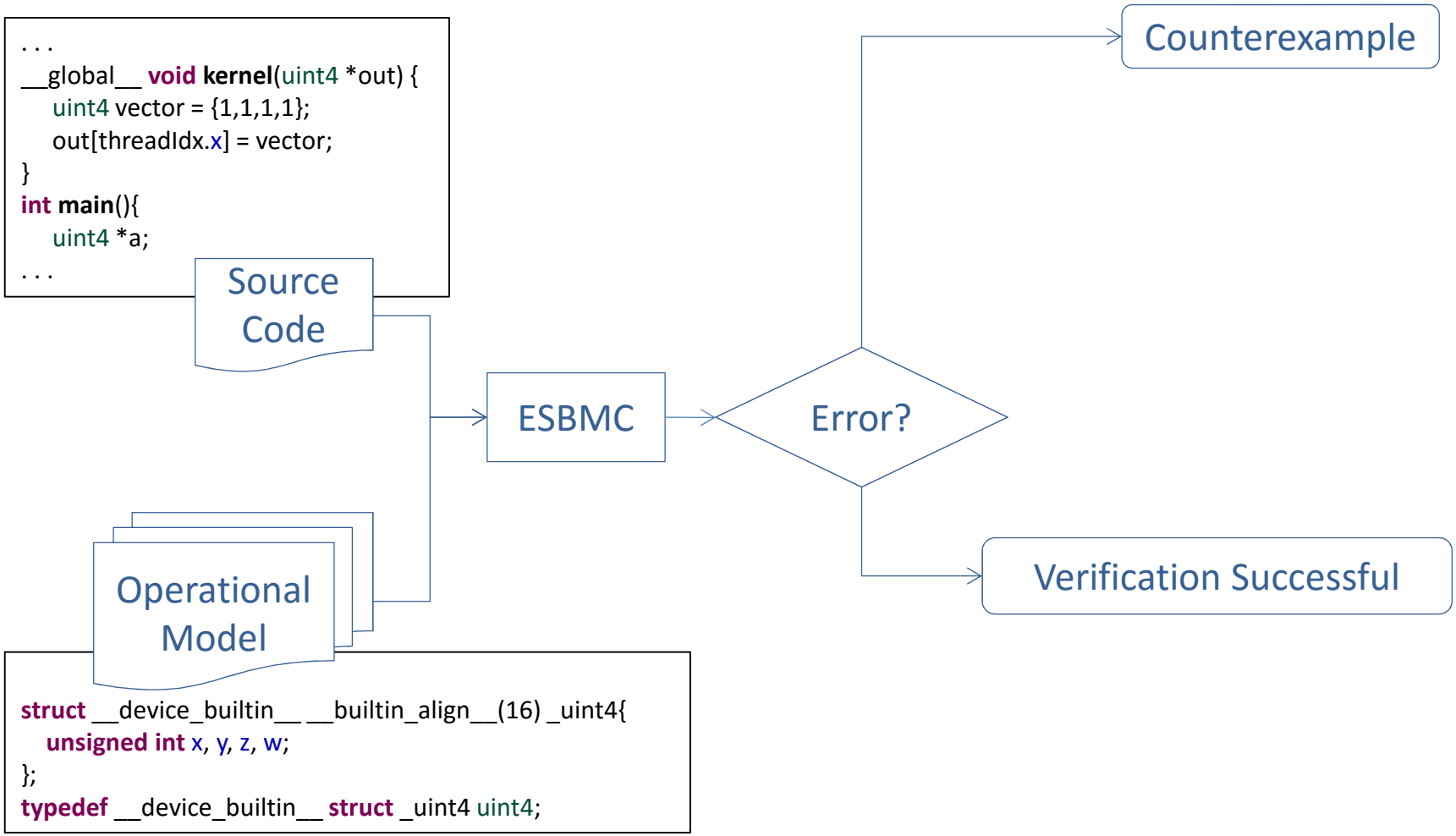
CUDA Operational Model (COM)

- COM aims to
  - **Abstractly** represent the associated CUDA libraries
    - checks **pre-** and **post-conditions**
    - simulates **behavior**
  - Reduce verification effort
    - by only checking **relevant behavior**
- COM allows ESBMC to check **specific properties** related to CUDA libraries
  - Other extensions to ESBMC based on operational models
    - ESBMC++ (ECBS'13) and ESBMC^{QtOM} (SPIN'16)
- CUDA is a **proprietary platform**
  - CUDA Programming Guide and IDE Nsight

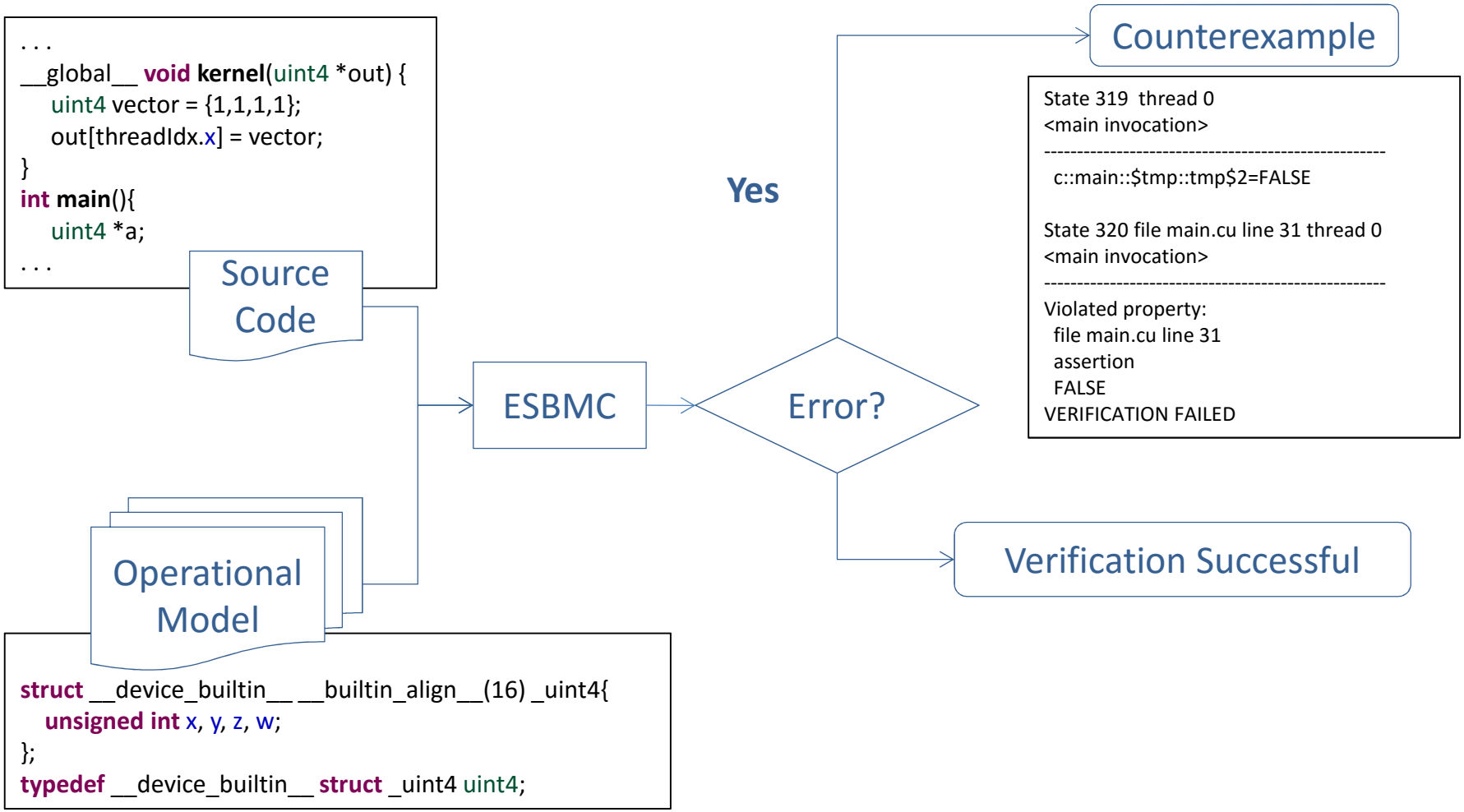
ESBMC-GPU: Verification Flow



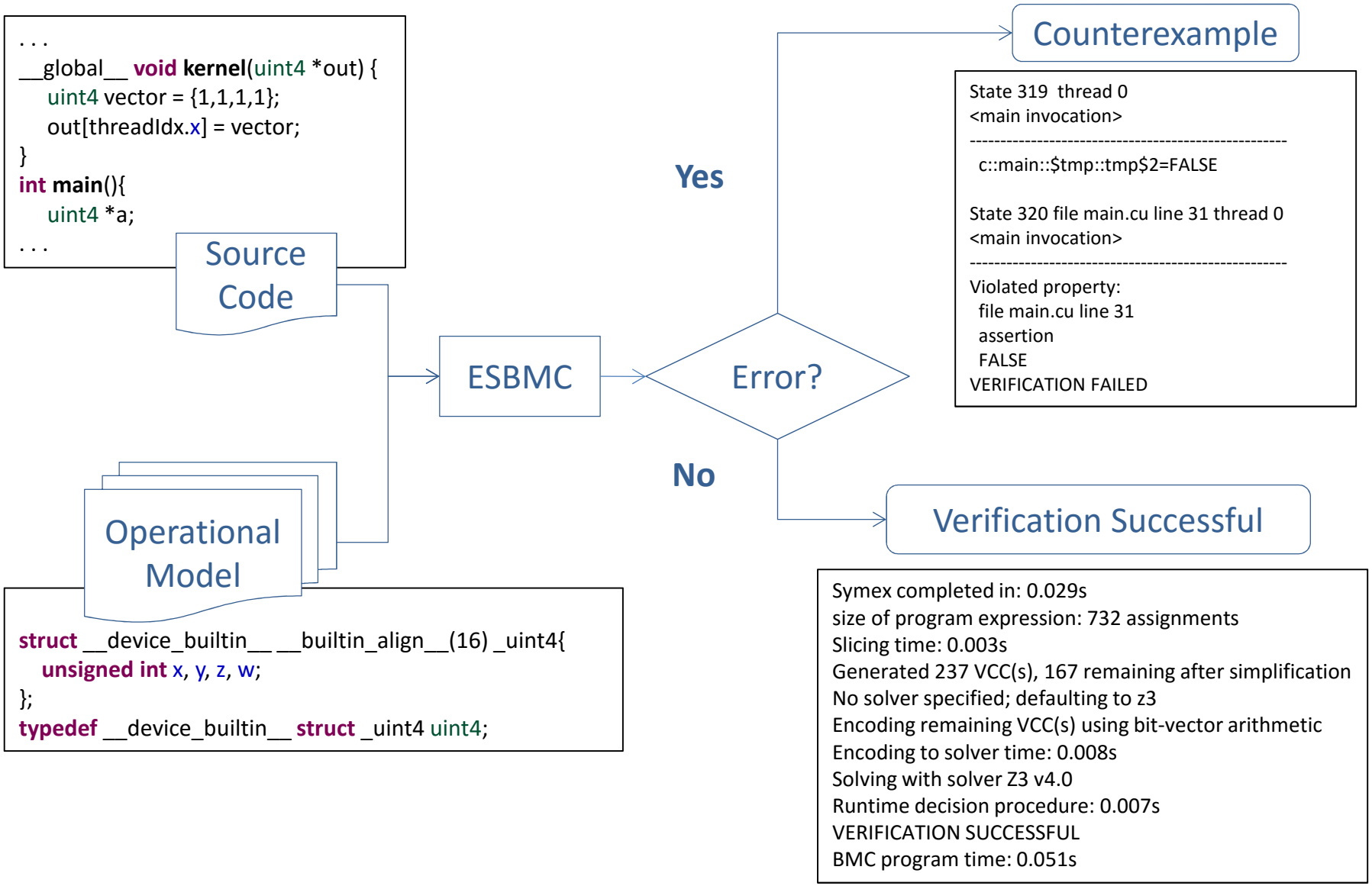
ESBMC-GPU: Verification Flow



ESBMC-GPU: Verification Flow



ESBMC-GPU: Verification Flow



COM Implementation: *cudaMalloc*

```
#include <cuda.h>
#include <stdio.h>
 #define N 2
 __global__ void definitions(int* A){
            atomicAdd(A,10);
}
 int main (){
            int a = 5;
            int *dev_a;
            cudaMalloc ((void**) &dev_a, sizeof(int));
            cudaMemcpy(dev_a, &a,
sizeof(int),cudaMemcpyHostToDevice);
            ESBMC_verify_kernel(definitions,1,N,dev_a);
            cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyD
eviceToHost);
            assert(a==25);
            cudaFree(dev_a);
            return 0;
}
```

COM Implementation: *cudaMalloc*

```
#include <cuda.h>
#include <stdio.h>
 #define N 2
 __global__ void definitions(int* A){
            atomicAdd(A,10);
}
 int main (){
            int a = 5;
            int *dev_a;
            cudaMalloc ((void**) &dev_a, sizeof(int));
            cudaMemcpy(dev_a, &a,
sizeof(int),cudaMemcpyHostToDevice);
            ESBMC_verify_kernel(definitions,1,N,dev_a);
            cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyD
eviceToHost);
            assert(a==25);
            cudaFree(dev_a);
            return 0;
}
```

COM Implementation: *cudaMalloc*

```
# cudaMalloc
cudaError_t cudaMalloc(void ** devPtr, size_t size) {
        cudaError_t tmp;
        __ESBMC_assert(size > 0, "Size to be allocated must be greater than zero");
        *devPtr = malloc(size);
         if (*devPtr == NULL) {
                tmp = CUDA_ERROR_OUT_OF_MEMORY;
                exit(1);
         } else {
                tmp = CUDA_SUCCESS;
         }
        __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
        lastError = tmp;
        return lastError;
}
```

pre-condition

COM Implementation: *cudaMalloc*

```
# cudaMalloc
cudaError_t cudaMalloc(void ** devPtr, size_t size) {
        cudaError_t tmp;
        __ESBMC_assert(size > 0, "Size to be allocated must be greater than zero");
        *devPtr = malloc(size);
        if (*devPtr == NULL) {
                tmp = CUDA_ERROR_OUT_OF_MEMORY;
                exit(1);
        } else {
                tmp = CUDA_SUCCESS;
        }
        __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
        lastError = tmp;
        return lastError;
}
```

simulate behavior

COM Implementation: *cudaMalloc*

```
# cudaMalloc
cudaError_t cudaMalloc(void ** devPtr, size_t size) {
        cudaError_t tmp;
        __ESBMC_assert(size > 0, "Size to be allocated must be greater than zero");
        *devPtr = malloc(size);
         if (*devPtr == NULL) {
                tmp = CUDA_ERROR_OUT_OF_MEMORY;
                exit(1);
         } else {
                tmp = CUDA_SUCCESS;
         }
        __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
        lastError = tmp;
        return lastError;
}
```

post-condition

Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library

Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library

CUDA program

```
_global_ void kernel(){
  A[tidx.x]=tidx.x;
}

int main(){
  int *a; int *dev_a;
  cudaMalloc(&dev_a,a,size);
  ...
  cudaMemcpy(dev_a,a,htd);
  ...
  ESBMC _verify_kernel(
kernel,M,N,dev_a);
  ...
  cudaMemcpy(a,dev_a,dth);
  ...
  cudaFree(dev_a);
  free(a);
}
```

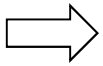
Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library

CUDA program

```
_global_ void kernel(){
  A[tidx.x]=tidx.x;
}

int main(){
  int *a; int *dev_a;
  cudaMalloc(&dev_a,a,size);
  ...
  cudaMemcpy(dev_a,a,htd);
  ...
  ESBMC _verify_kernel(
kernel,M,N,dev_a);
  ...
  cudaMemcpy(a,dev_a,dth);
  ...
  cudaFree(dev_a);
  free(a);
}
```



COM

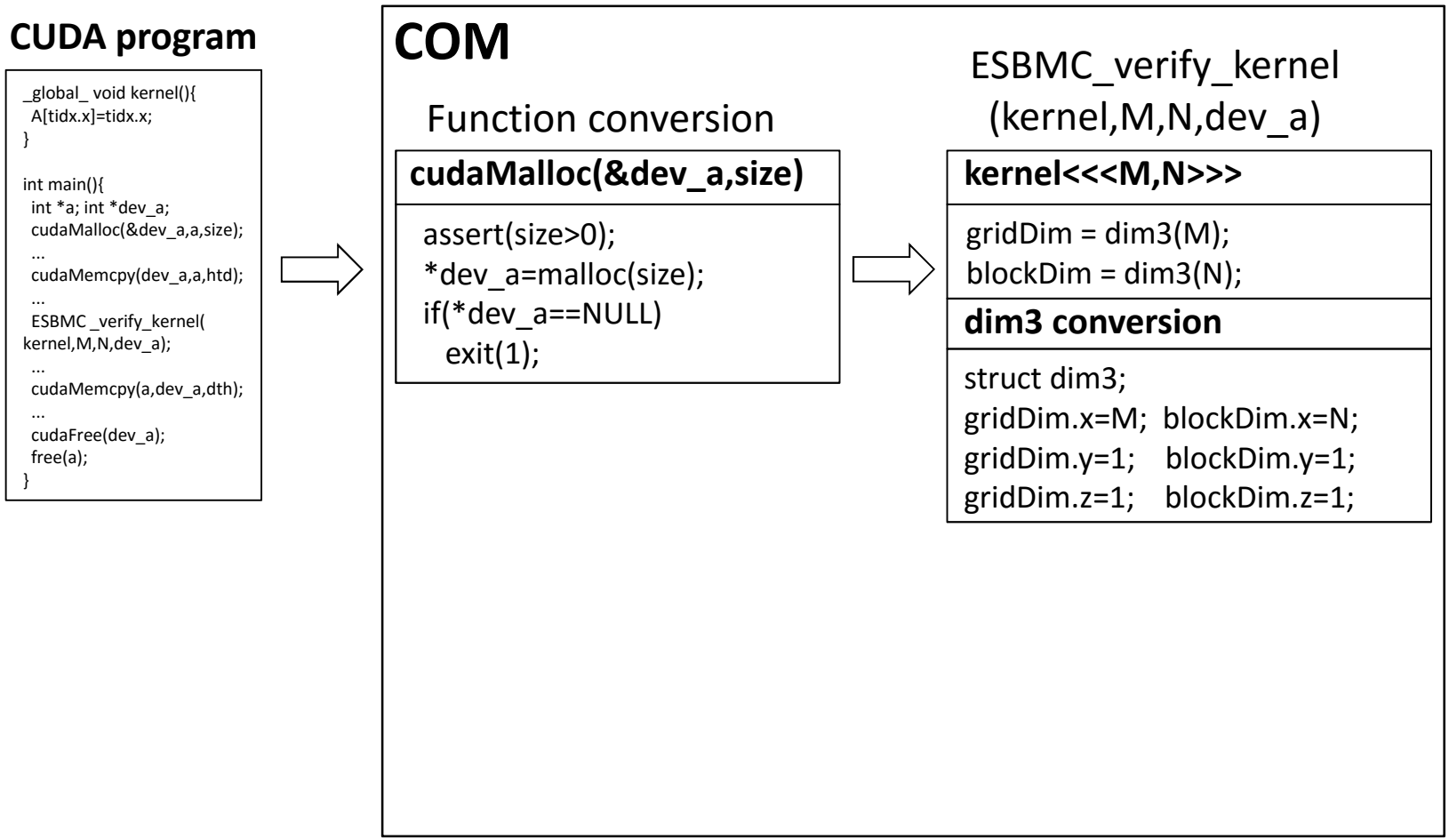
Function conversion

cudaMalloc(&dev_a,size)

```
assert(size>0);
*dev_a=malloc(size);
if(*dev_a==NULL)
  exit(1);
```

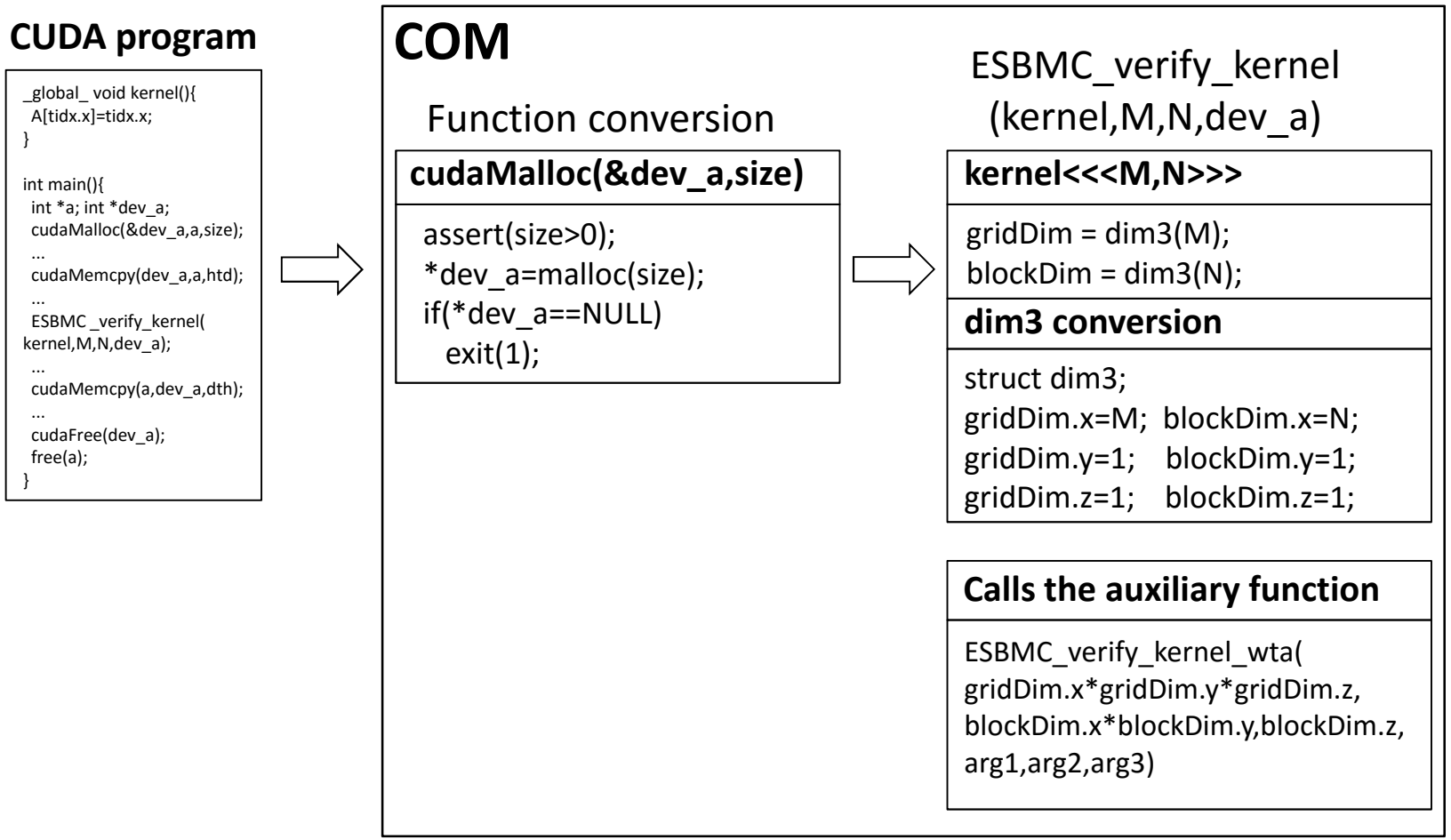
Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library



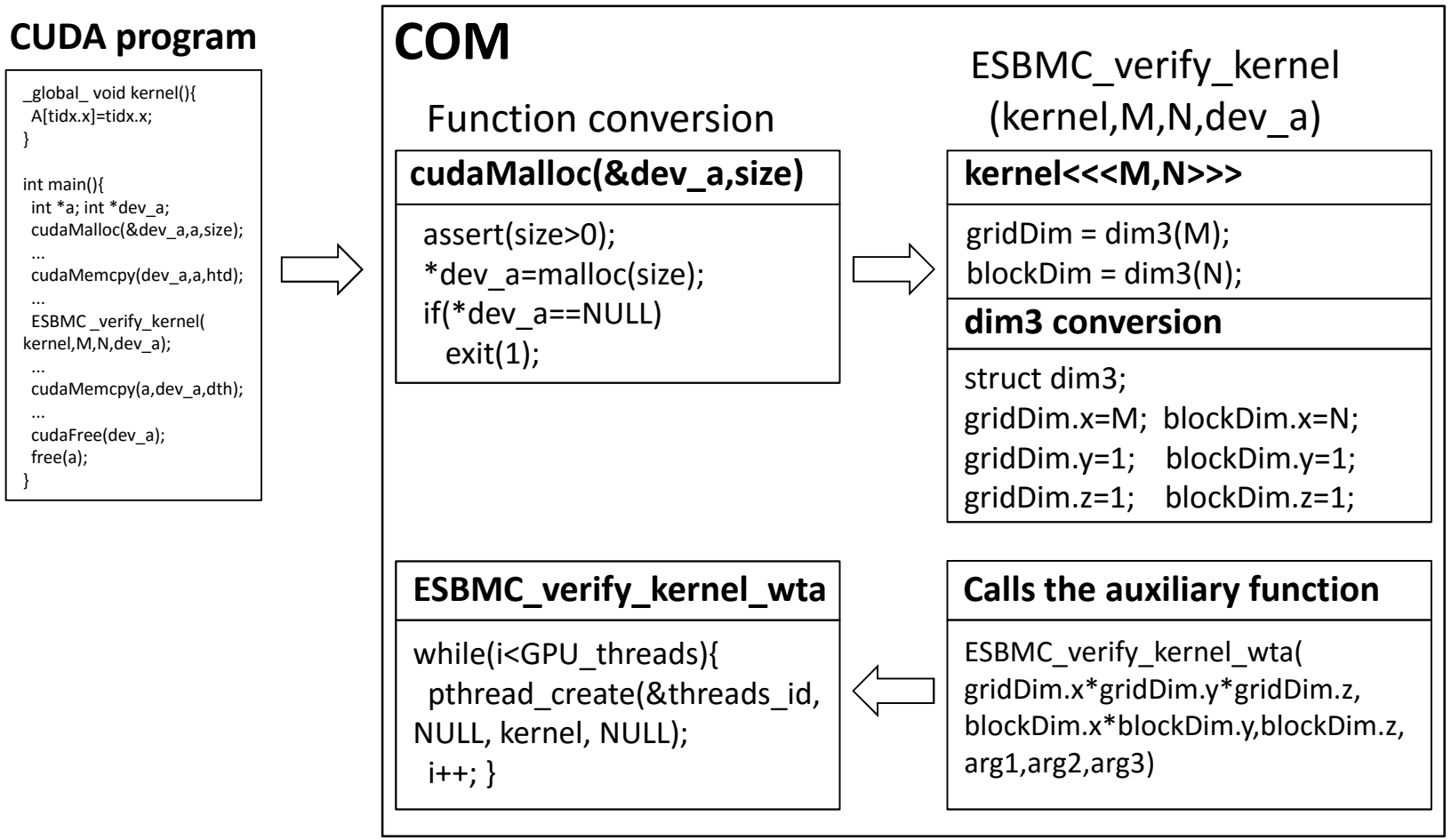
Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library



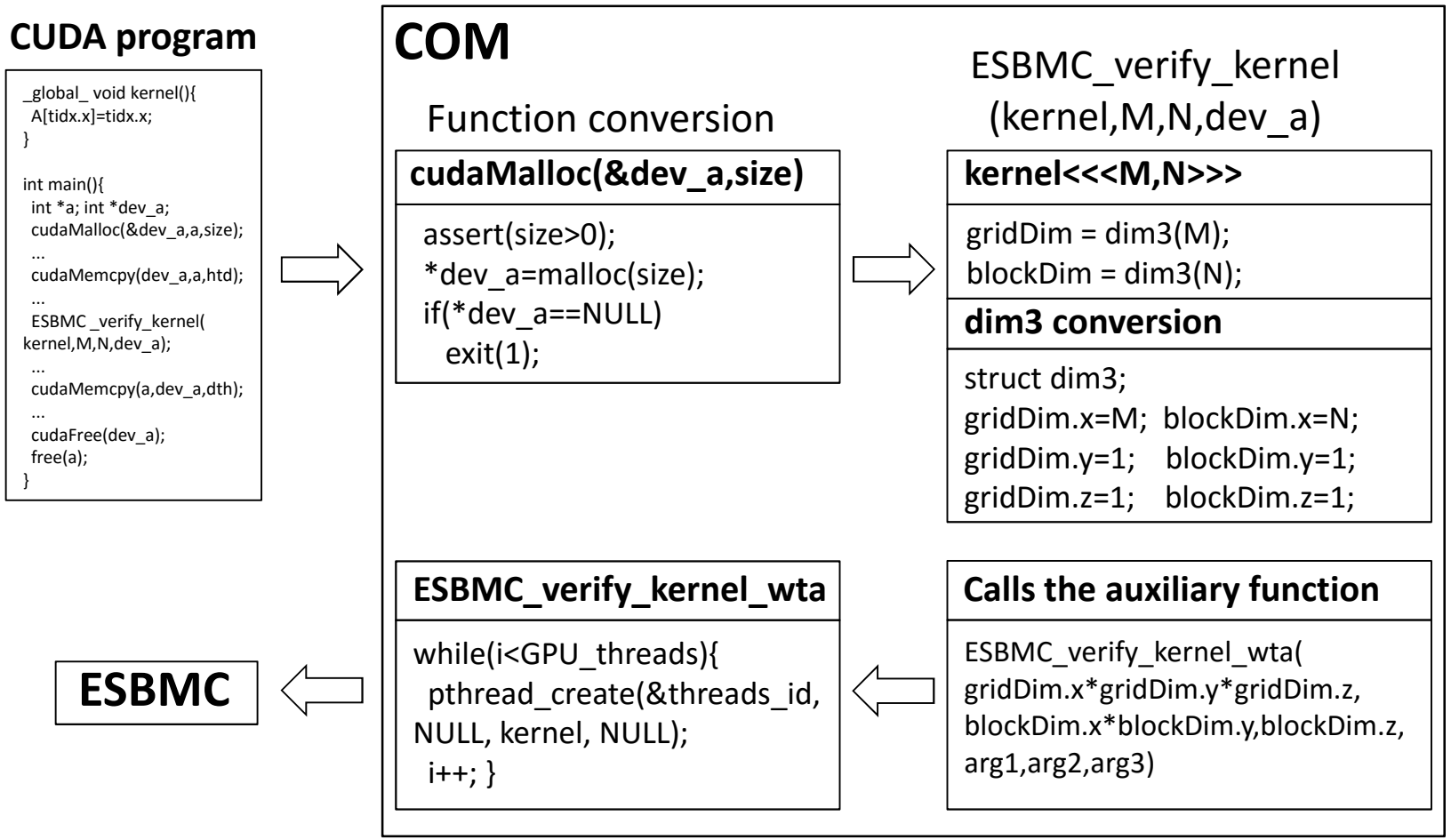
Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library



Modeling Kernels with Pthreads in COM

- Verification model adopts the CPU parallel processing
  - Using the Pthread/POSIX library



Monotonic Partial Order Reduction (MPOR)

- MPOR classifies thread transitions in a multi-threaded program
  - Each transition may be dependent or independent
    - Identify interleaving pairs which result in the same state

Monotonic Partial Order Reduction (MPOR)

- MPOR classifies thread transitions in a multi-threaded program
  - Each transition may be dependent or independent
    - Identify interleaving pairs which result in the same state
- First application of the technique to verify CUDA-based programs
  - Reduction in time and verification effort
  - Elimination of threads interleavings that **access different array positions**

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

1. **function** MPOR (ν, π)
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return “independent” on π and terminates
6. Return “dependent” on π and terminates
7. **end function**

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0, \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
    a[threadIdx.x] = threadIdx.x ;
}
```

$v_0: t_0, 0, a[0] = 0\ , a[1] = 0$

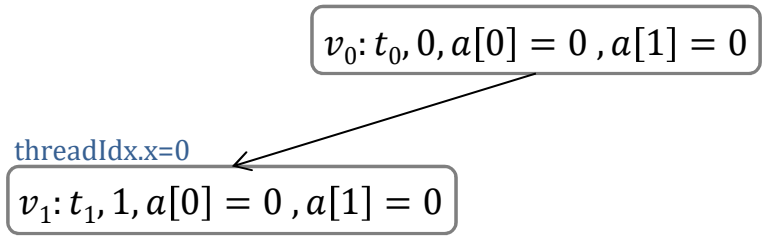
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
  a[threadIdx.x] = threadIdx.x ;
}
```



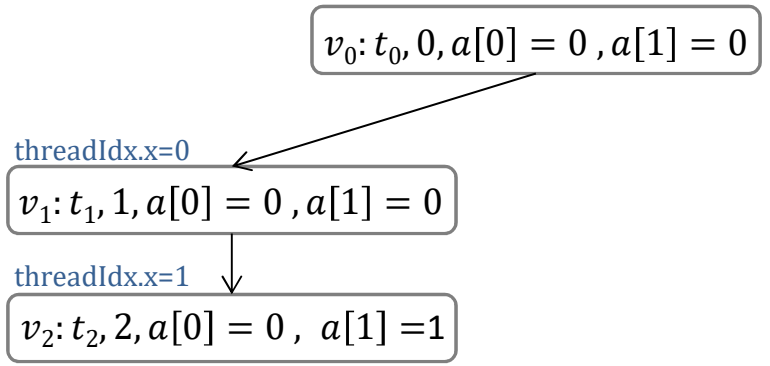
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
  a[threadIdx.x] = threadIdx.x ;
}
```



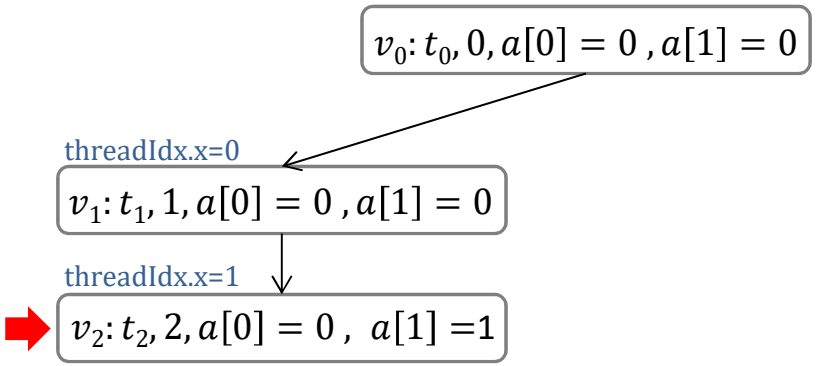
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
    a[threadIdx.x] = threadIdx.x ;
}
```



MPOR Applied to CUDA-based Programs

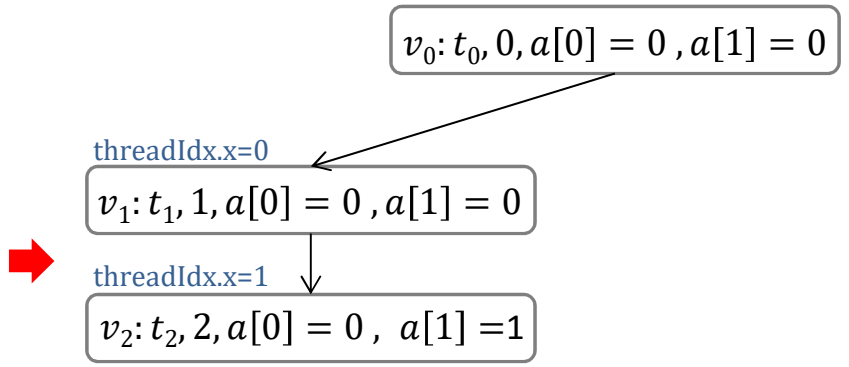
- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
    a[threadIdx.x] = threadIdx.x ;
}
```

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

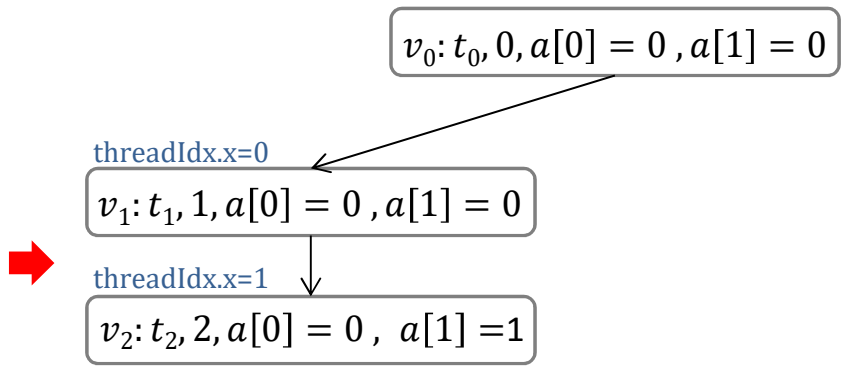
v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. ➡ Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
   a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

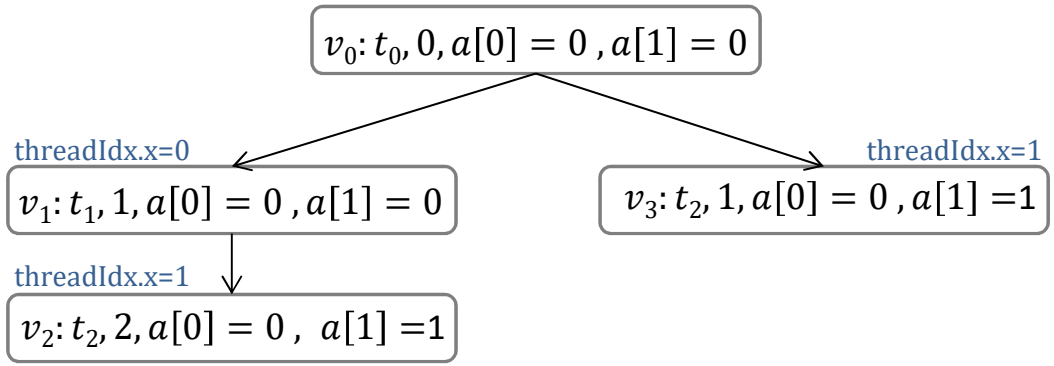
v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
  a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

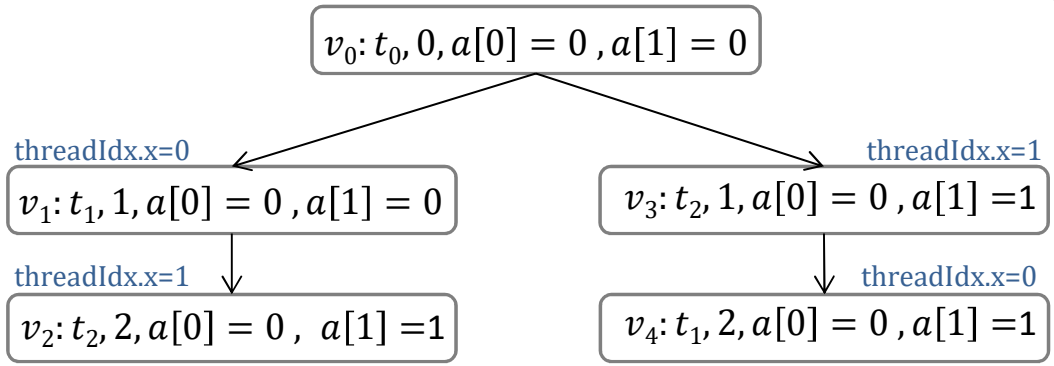
v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
  a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

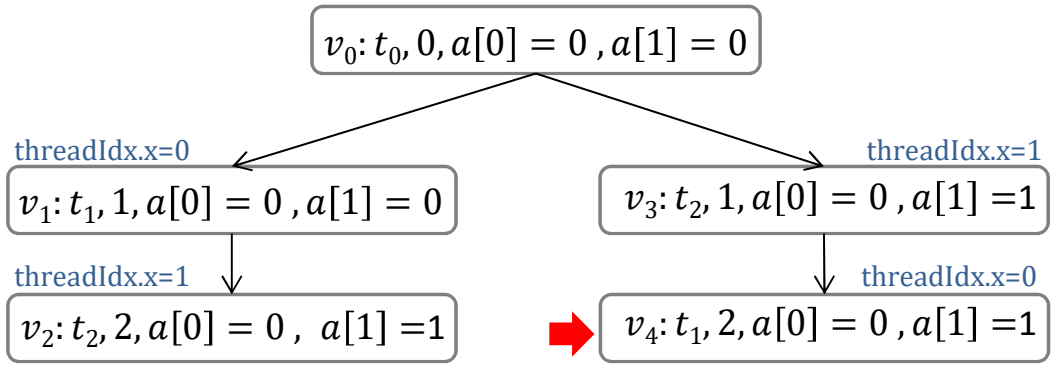
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
   a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

$t_2 \rightarrow t_1$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

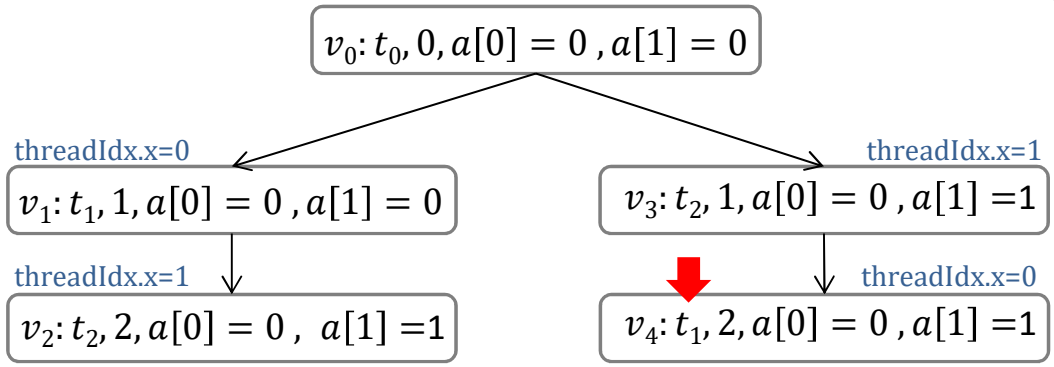
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
   a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

$t_2 \rightarrow t_1$
s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

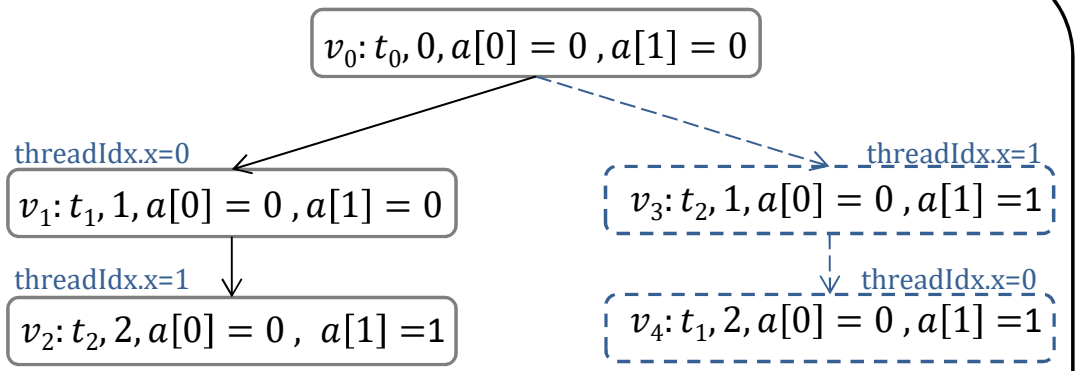
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. ➡ Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel1(int *a){
   a[threadIdx.x] = threadIdx.x ;
}
```

Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

$t_2 \rightarrow t_1$
s_2: a[0] = 0
a[1] = 1



$t_1 \rightarrow t_2$ or $t_2 \rightarrow t_1$ result in the same state, this is an independent transition

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
    if(a[1]==1)
       a[threadIdx.x+2] = threadIdx.x ;
    else
       a[threadIdx.x] = threadIdx.x;
```

$v_0: t_0, 0, a[0] = 0\ , a[1] = 0$

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
    a[threadIdx.x+2] = threadIdx.x ;
  else
    a[threadIdx.x] = threadIdx.x;
```

threadIdx.x=0

$v_0: t_0, 0, a[0] = 0, a[1] = 0$

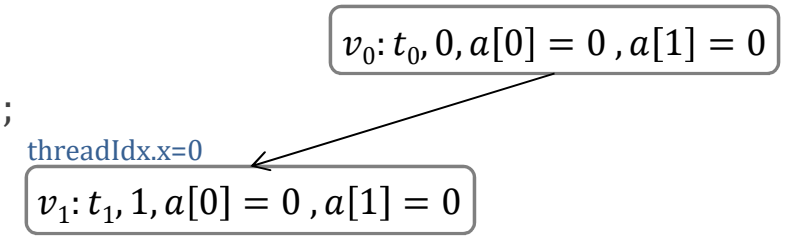
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```



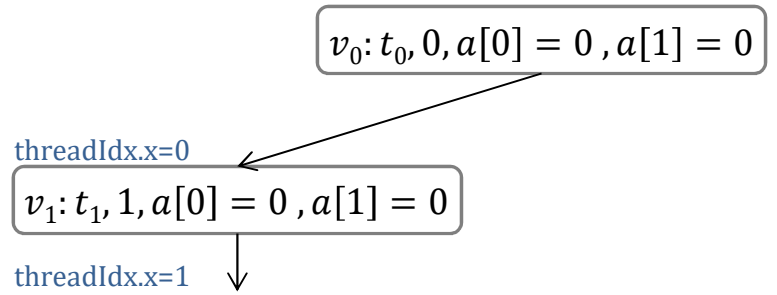
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
    a[threadIdx.x+2] = threadIdx.x ;
  else
    a[threadIdx.x] = threadIdx.x;
```



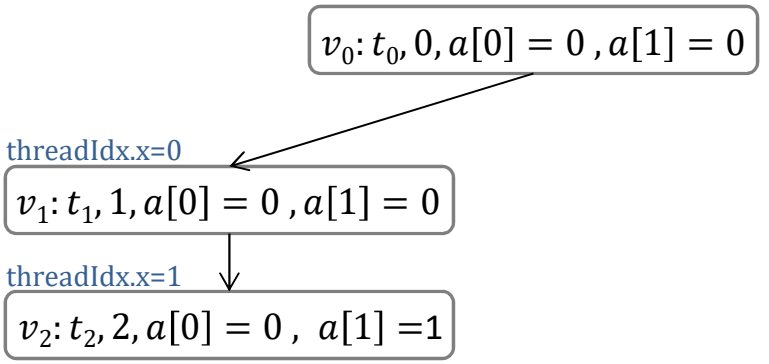
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```



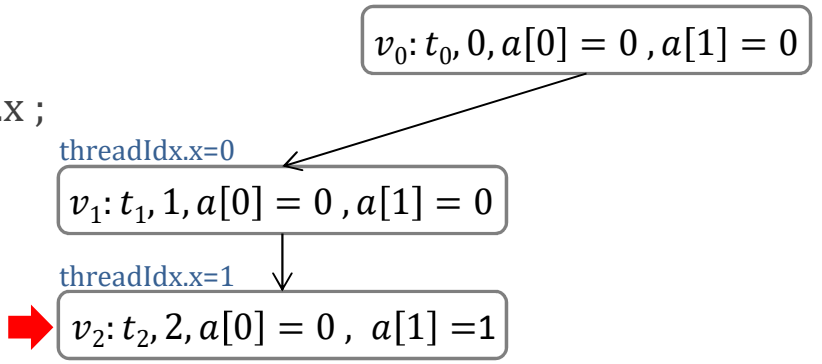
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
    if(a[1]==1)
       a[threadIdx.x+2] = threadIdx.x ;
    else
       a[threadIdx.x] = threadIdx.x;
```



$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

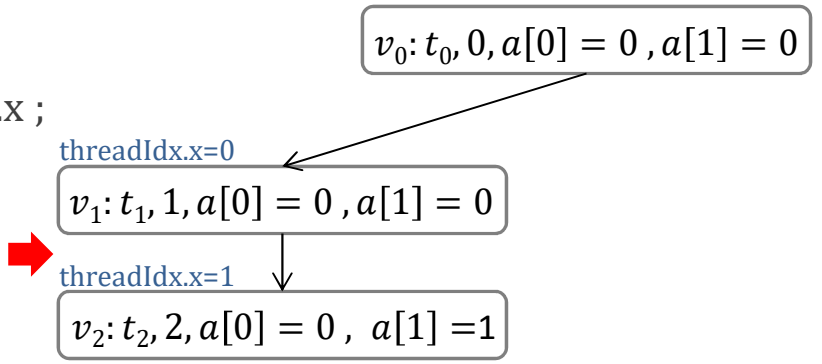
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
    if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
    else
      a[threadIdx.x] = threadIdx.x;
```



$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

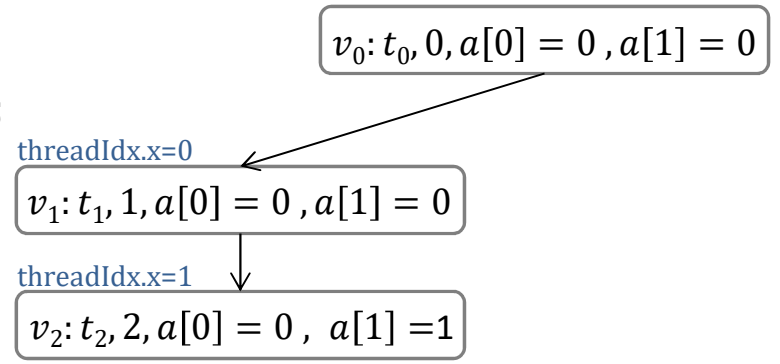
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```



Dependent

$t_1 \rightarrow t_2$
s_2: a[0] = 0
a[1] = 1

MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

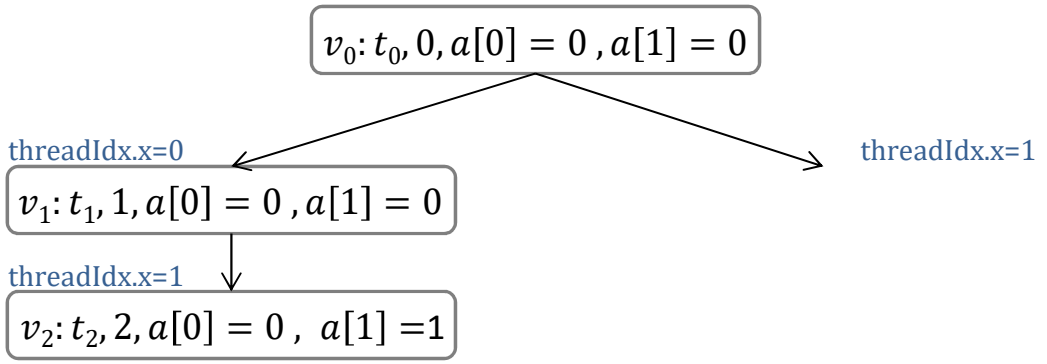
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
     a[threadIdx.x+2] = threadIdx.x ;
  else
     a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$

s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

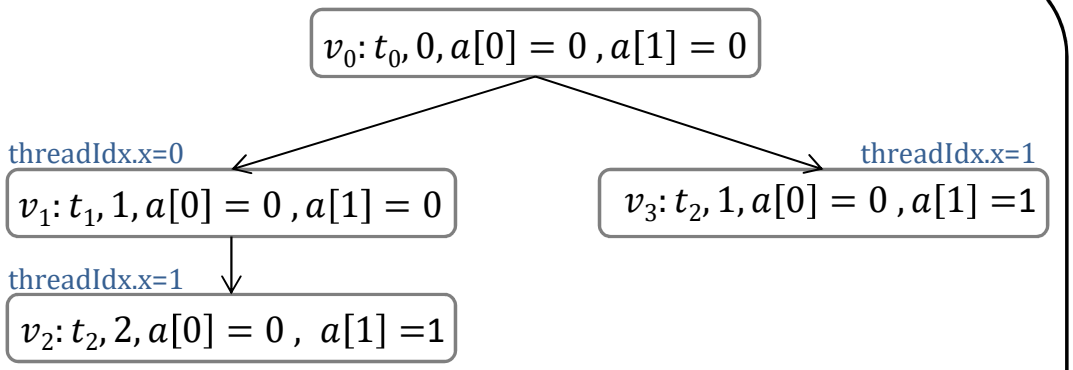
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$

s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

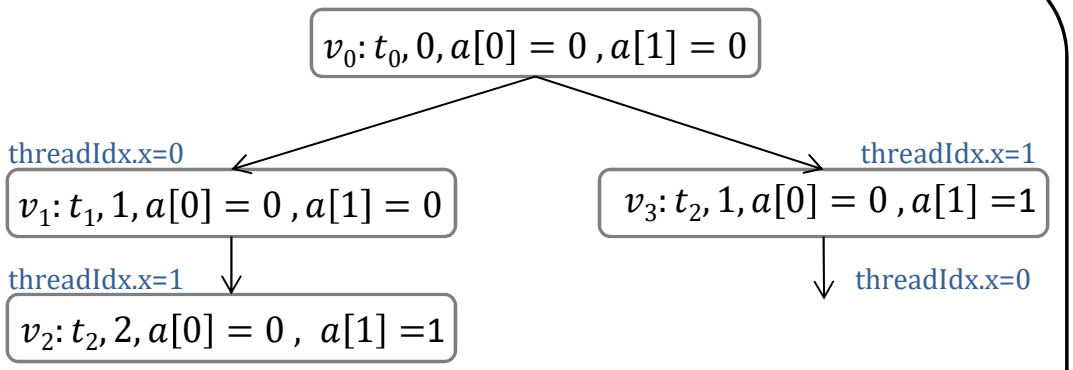
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
     a[threadIdx.x+2] = threadIdx.x ;
  else
     a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$

s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

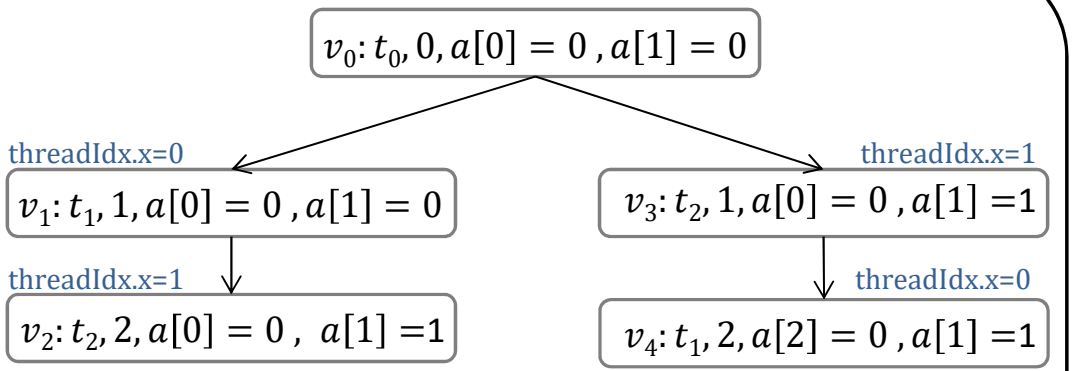
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
    a[threadIdx.x+2] = threadIdx.x ;
  else
    a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$

s_2: a[0] = 0
a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$

2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates

7. **end function**

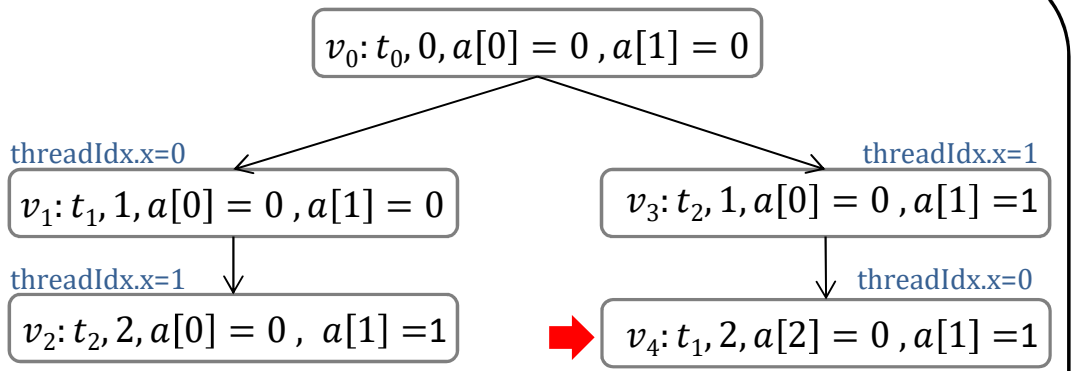
v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$: s_2: a[0] = 0, a[1] = 1

$t_2 \rightarrow t_1$: s_3: a[2] = 0, a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

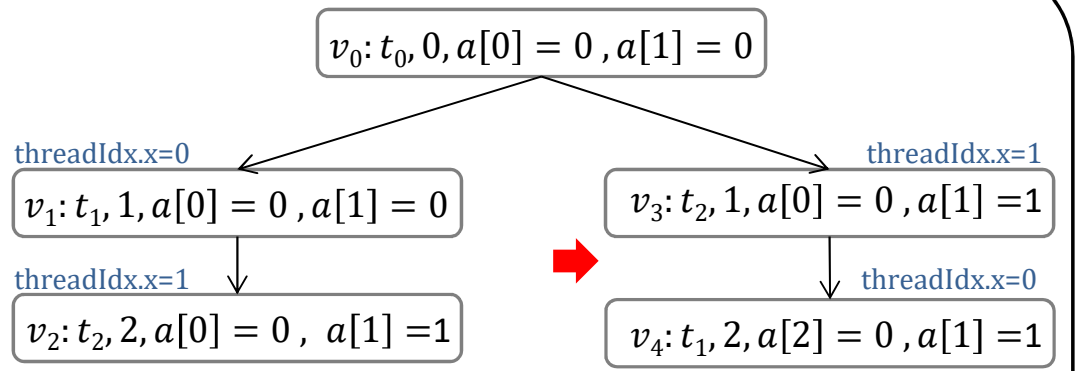
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
   if(a[1]==1)
      a[threadIdx.x+2] = threadIdx.x ;
   else
      a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$: s_2: a[0] = 0, a[1] = 1

$t_2 \rightarrow t_1$: s_3: a[2] = 0, a[1] = 1



MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

v = (A_i, C_i, s_i)
A_i: active thread
C_i: context switch
s_i: current state

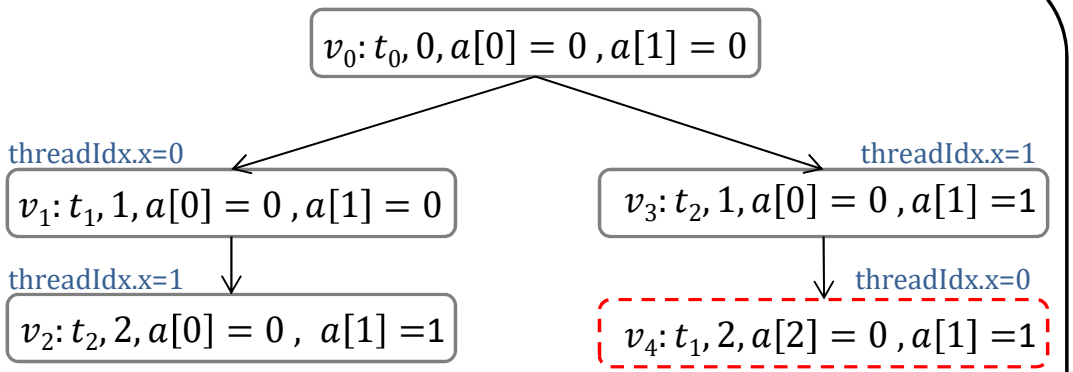
1. **function** MPOR (v, π) $\pi = \{v_0 \ldots, v_k\}$
2. Check whether s_i exists in π; otherwise, go to step 4
3. Check whether A_i produces a new state in π; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on π; otherwise, go to step 6
5. Return "independent" on π and terminates
6. Return "dependent" on π and terminates
7. **end function**

```
kernel (int *a)
  if(a[1]==1)
    a[threadIdx.x+2] = threadIdx.x ;
  else
    a[threadIdx.x] = threadIdx.x;
```

Dependent

$t_1 \rightarrow t_2$: s_2: a[0] = 0, a[1] = 1

$t_2 \rightarrow t_1$: s_3: a[2] = 0, a[1] = 1



$t_1 \rightarrow t_2$ computes a different state of $t_2 \rightarrow t_1$, resulting in dependent transitions

Two-threads Analysis

- Reduction for two-threads during the program verification

Two-threads Analysis

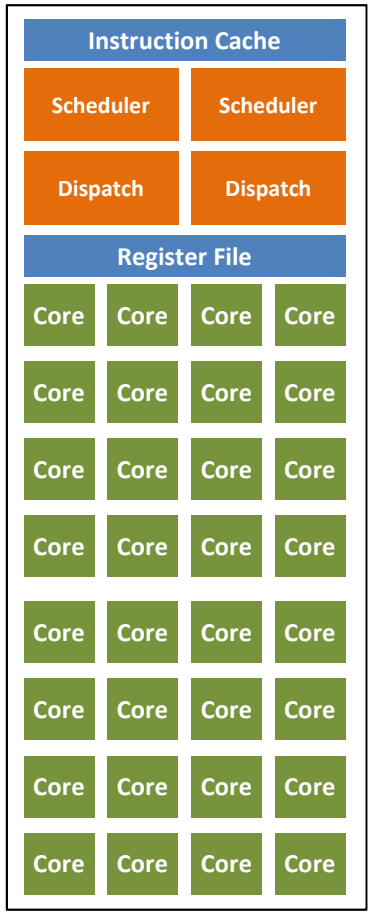
- Reduction for two-threads during the program verification
  - If an error is found between 2 threads in a block, it will also be found for more threads
    - ➢ This proposition holds due to the GPU architecture

Two-threads Analysis

- Reduction for two-threads during the program verification
  - If an error is found between 2 threads in a block, it will also be found for more threads
    - ➢ This proposition holds due to the GPU architecture
  - This technique is also used by other GPU kernel verification tools (*e.g.,* GPUVerify and PUG)

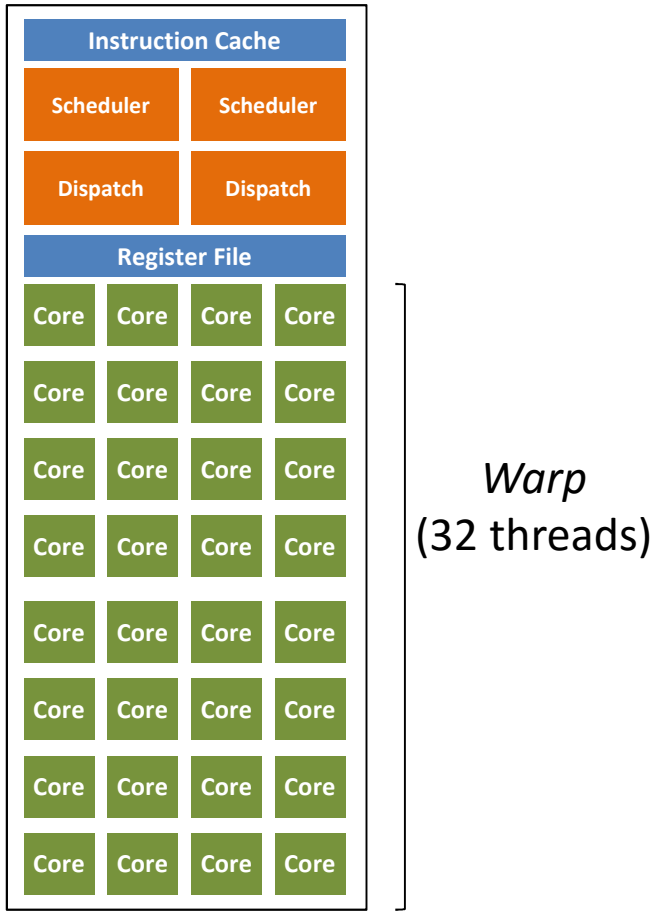
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor



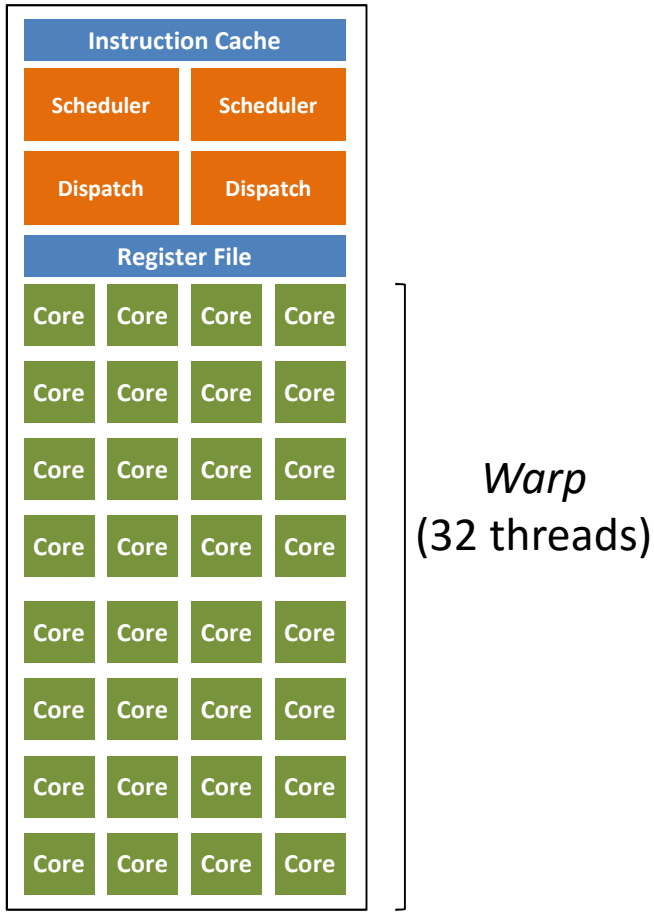
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

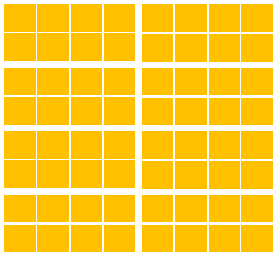


Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor



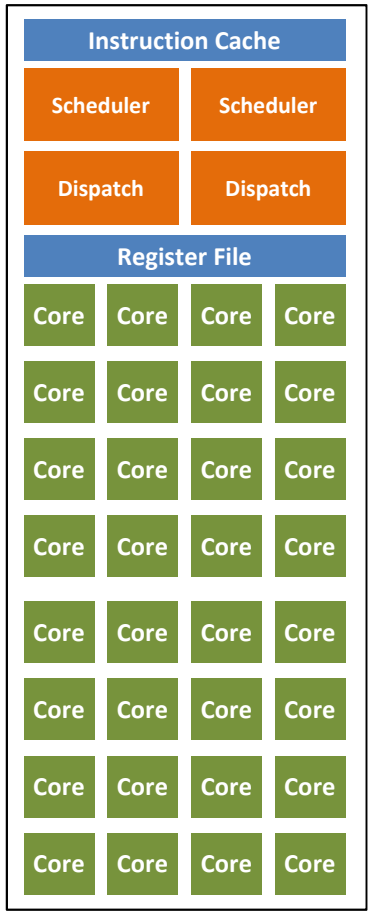
Block of threads
(64 threads)



Warp
(32 threads)

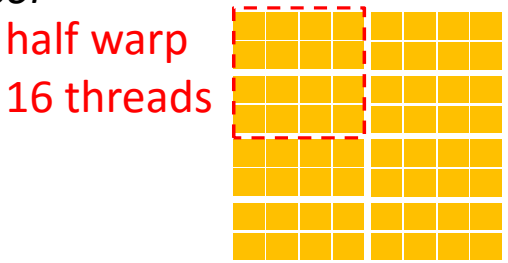
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor



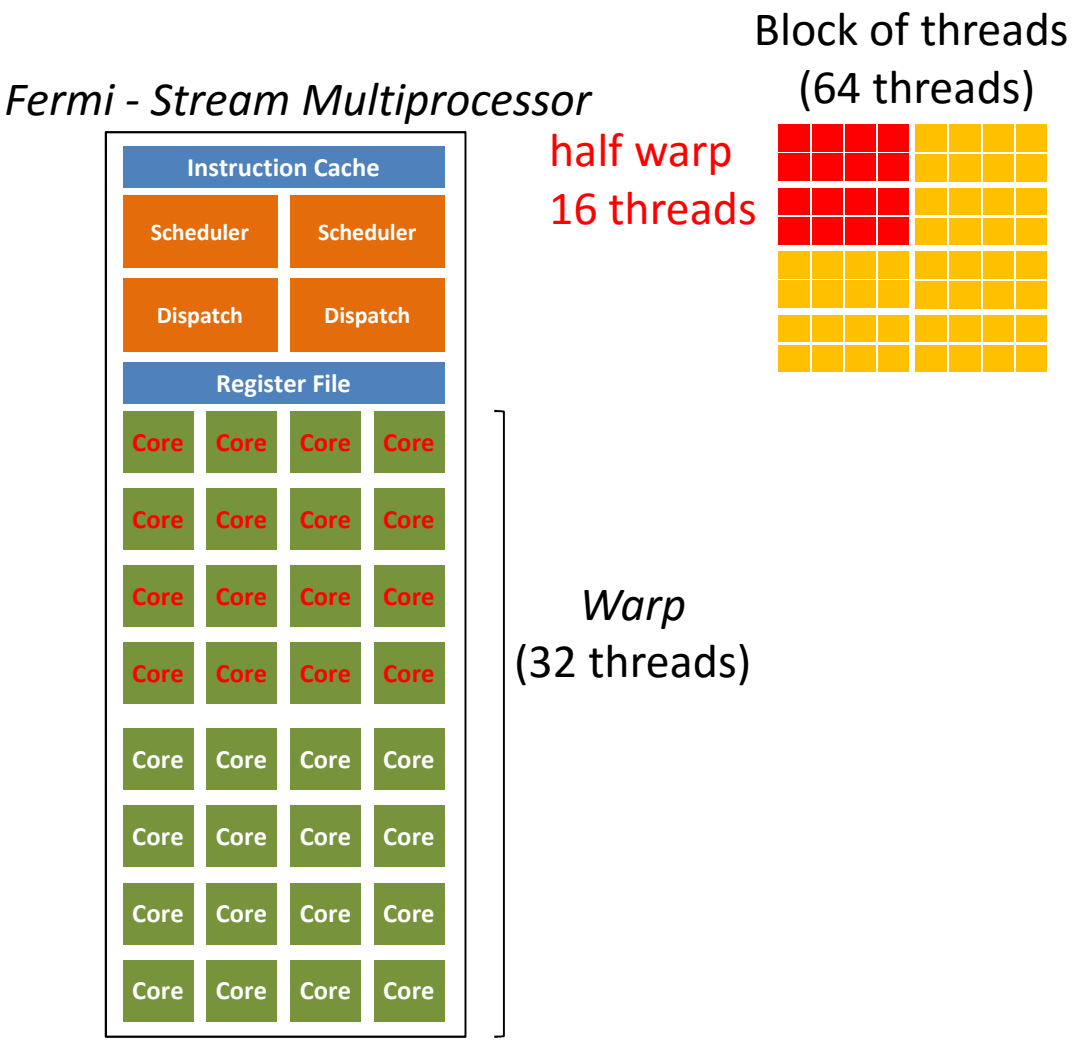
Warp
(32 threads)

Block of threads
(64 threads)

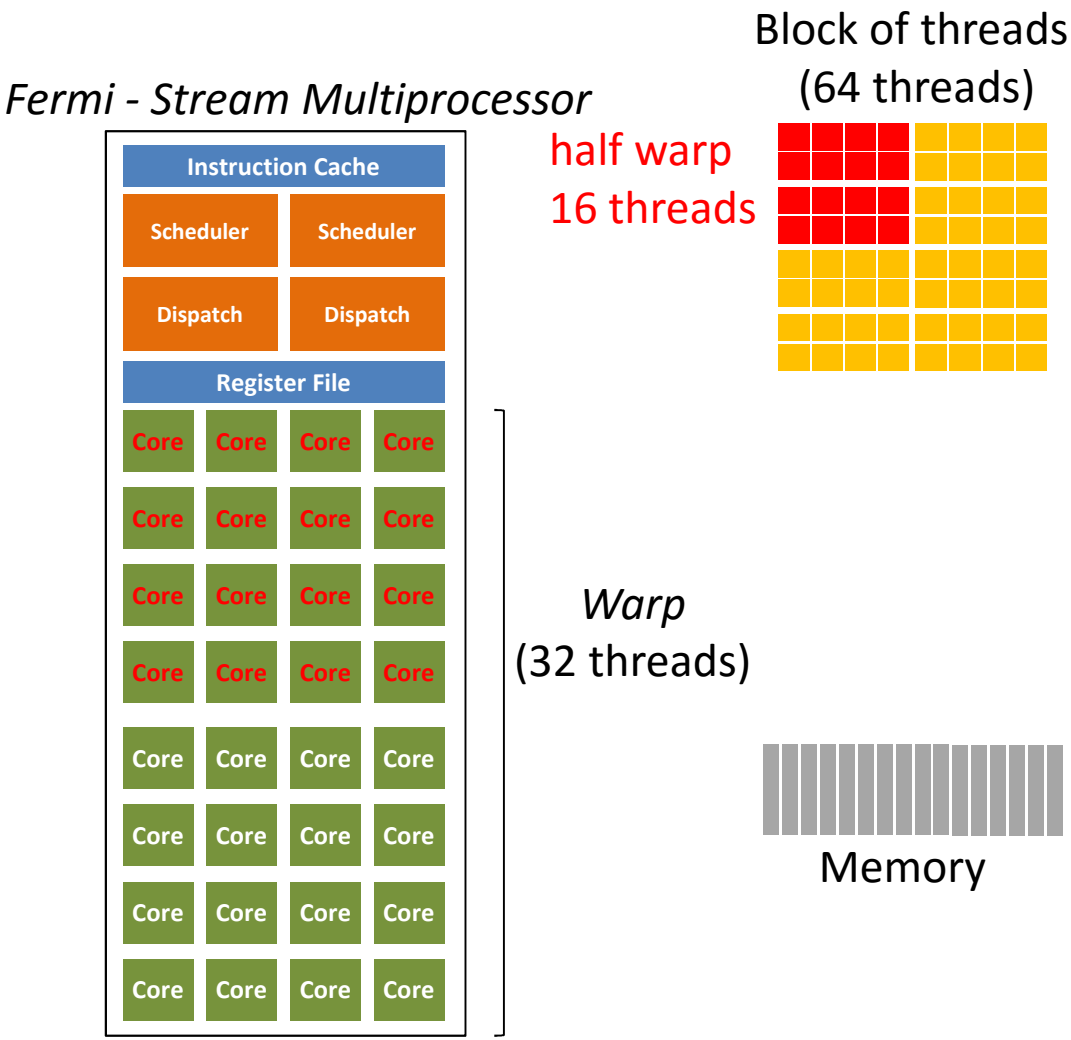


One thread group is processed by a half warp in the SM

Two-threads Analysis in Fermi

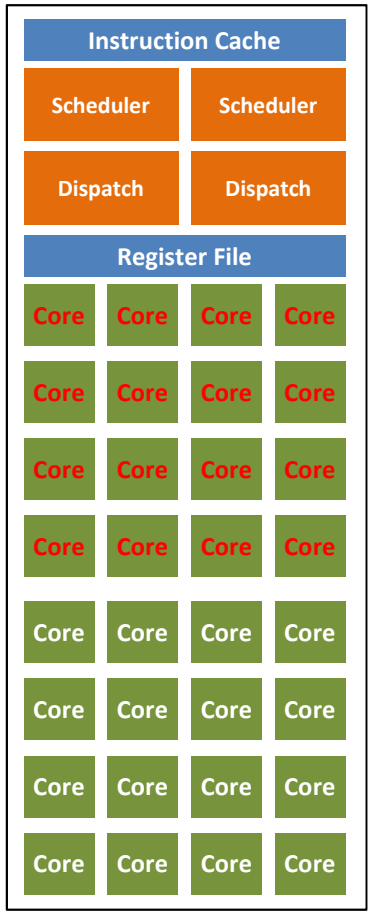


Two-threads Analysis in Fermi

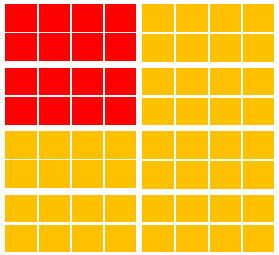


Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

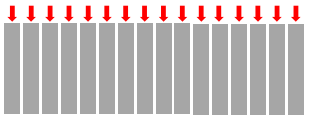


Block of threads
(64 threads)



half warp
16 threads

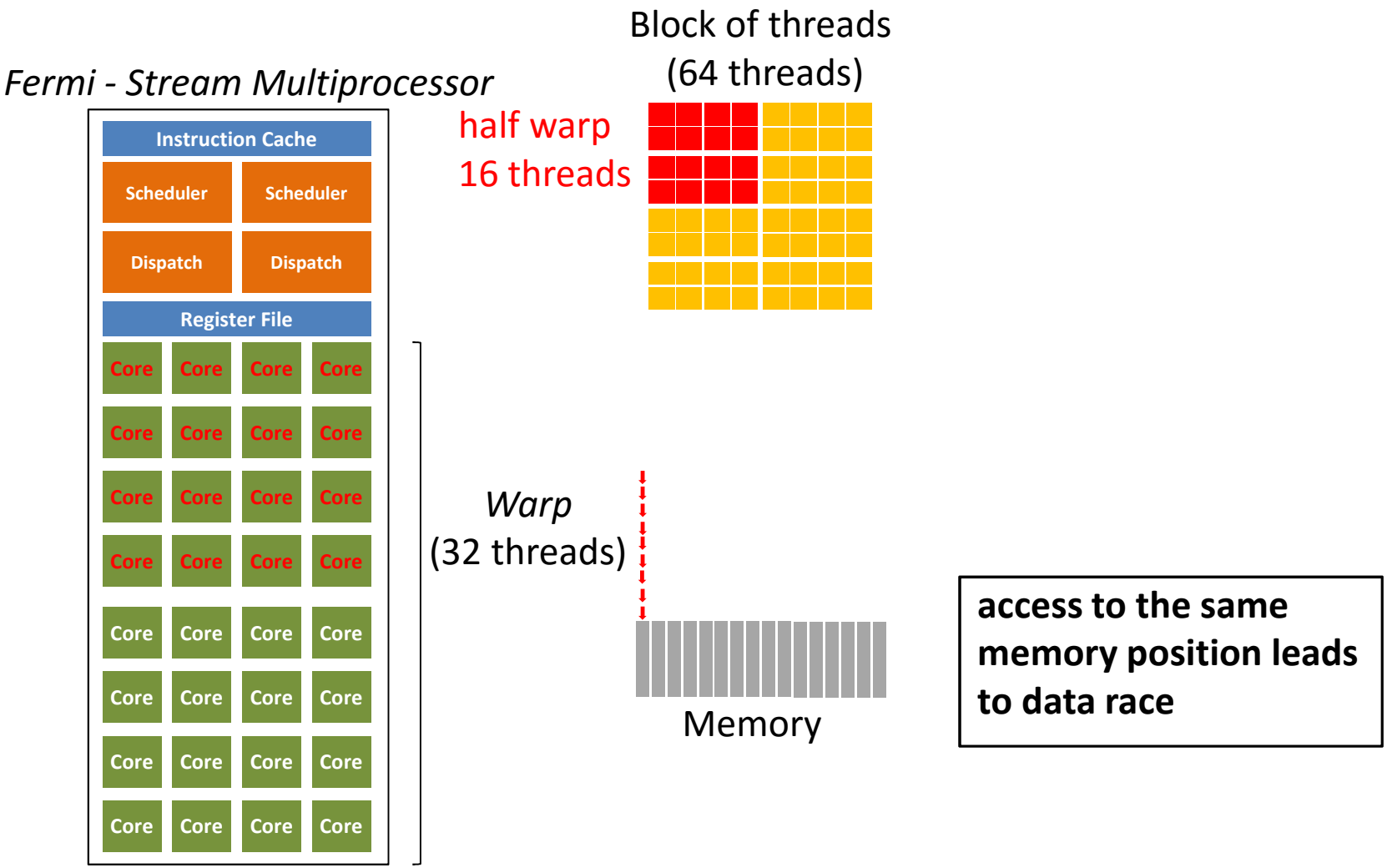
Warp
(32 threads)



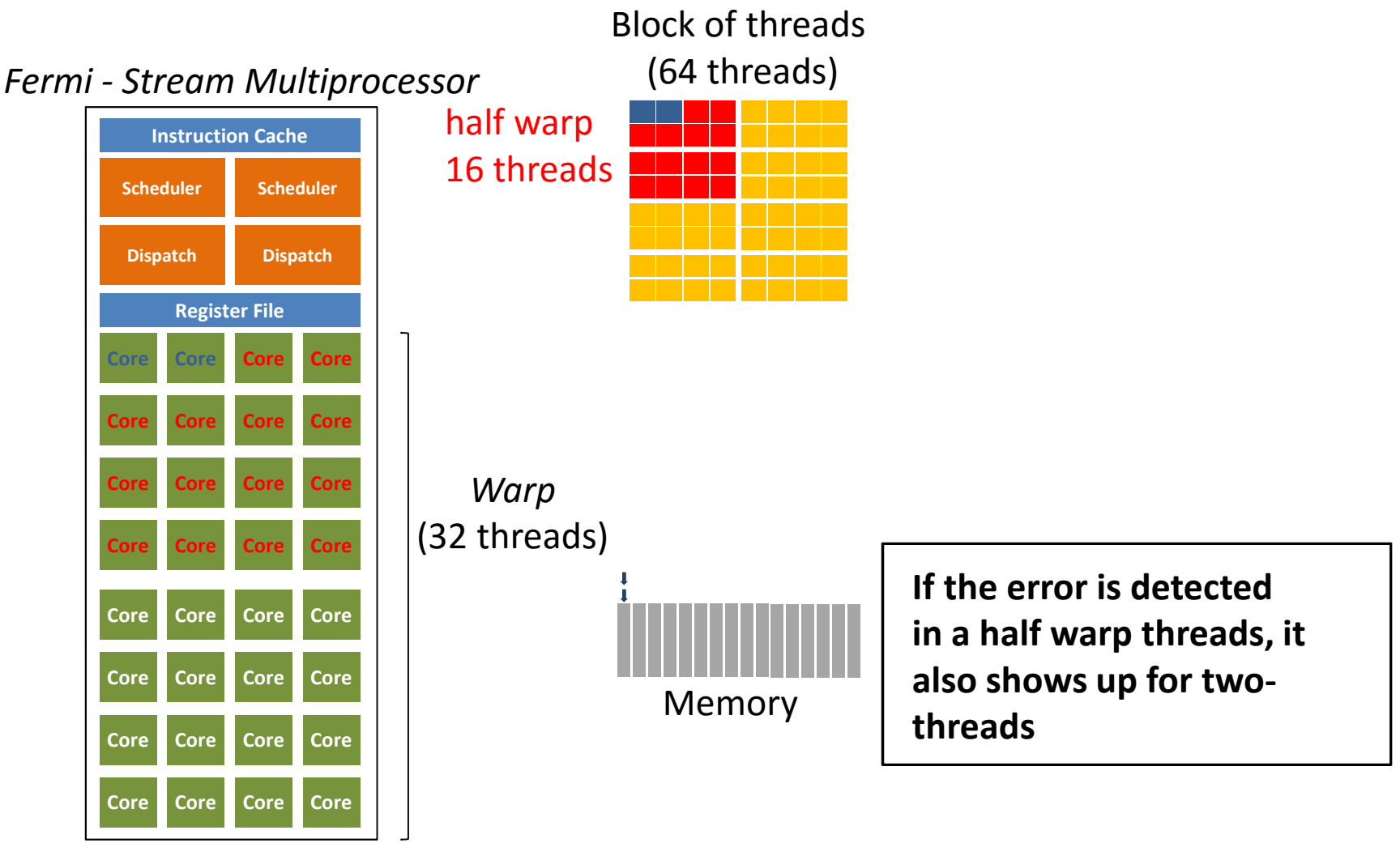
Memory

there is no data race to access different memory positions

Two-threads Analysis in Fermi



Two-threads Analysis in Fermi



Experimental Evaluation

- **Objective:** check whether ESBMC-GPU is able to correctly verify CUDA-based programs

Experimental Evaluation

- **Objective:** check whether ESBMC-GPU is able to correctly verify CUDA-based programs
  - Ensure that verification results are correct according to the CUDA specification

Experimental Evaluation

- **Objective:** check whether ESBMC-GPU is able to correctly verify CUDA-based programs
  - Ensure that verification results are correct according to the CUDA specification
- Research Questions (RQ)
  - **RQ1 (sanity check)** which results does ESBMC-GPU obtain upon verifying benchmarks that compose the specified suite?

Experimental Evaluation

- **Objective:** check whether ESBMC-GPU is able to correctly verify CUDA-based programs
  - Ensure that verification results are correct according to the CUDA specification
- Research Questions (RQ)
  - **RQ1 (sanity check)** which results does ESBMC-GPU obtain upon verifying benchmarks that compose the specified suite?
  - **RQ2 (comparison with other tools)** what is ESBMC-GPU performance if compared to GKLEE, GPUVerify, PUG, and CIVL?

Experimental Evaluation

- **Objective:** check whether ESBMC-GPU is able to correctly verify CUDA-based programs
  - Ensure that verification results are correct according to the CUDA specification
- Research Questions (RQ)
  - **RQ1 (sanity check)** which results does ESBMC-GPU obtain upon verifying benchmarks that compose the specified suite?
  - **RQ2 (comparison with other tools)** what is ESBMC-GPU performance if compared to GKLEE, GPUVerify, PUG, and CIVL?
- Standard PC desktop, time-out 900 seconds

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)
  - CUDA libraries (*e.g., curand.h*)

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)
  - CUDA libraries (*e.g., curand.h*)
  - Data types *int*, *float*, *char,* and its modifiers (*long and unsigned*)

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)
  - CUDA libraries (*e.g., curand.h*)
  - Data types *int*, *float*, *char,* and its modifiers (*long and unsigned*)
  - Pointers to variables and functions

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)
  - CUDA libraries (*e.g., curand.h*)
  - Data types *int*, *float*, *char,* and its modifiers (*long and unsigned*)
  - Pointers to variables and functions
  - Typedefs

CUDA Benchmarks

- Extracted 154 benchmarks from the literature
  - Arithmetic operations
  - Device functions call
  - Specific functions of:
    - C/C++ (*e.g., memset, assert*)
    - CUDA (*e.g., atomicAdd, cudaMemcpy, cudaMalloc, cudaFree, syncthreads*)
  - CUDA libraries (*e.g., curand.h*)
  - Data types *int*, *float*, *char,* and its modifiers (*long and unsigned*)
  - Pointers to variables and functions
  - Typedefs
  - CUDA intrinsic variables (*e.g., uint4*)

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function
- **PUG** checks data races, barrier synchronization, and conflicts with shared memory

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function
- **PUG** checks data races, barrier synchronization, and conflicts with shared memory
- **GKLEE** is based on concrete and symbolic execution
  - Supports the verification of barriers synchronization and race condition

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function
- **PUG** checks data races, barrier synchronization, and conflicts with shared memory
- **GKLEE** is based on concrete and symbolic execution
  - Supports the verification of barriers synchronization and race condition
- **CIVL** is a framework for static analysis and concurrent program verification

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function
- **PUG** checks data races, barrier synchronization, and conflicts with shared memory
- **GKLEE** is based on concrete and symbolic execution
  - Supports the verification of barriers synchronization and race condition
- **CIVL** is a framework for static analysis and concurrent program verification
  - supports MPI, POSIX, OpenMP, CUDA, and C11

State-of-the-Art GPU Verifiers

- **GPUVerify** checks data race and barrier divergence
  - There is no support for the main function
- **PUG** checks data races, barrier synchronization, and conflicts with shared memory
- **GKLEE** is based on concrete and symbolic execution
  - Supports the verification of barriers synchronization and race condition
- **CIVL** is a framework for static analysis and concurrent program verification
  - supports MPI, POSIX, OpenMP, CUDA, and C11
  - symbolic execution, POR, and GPU threads with Pthread

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	7	8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
			30	15	24
			9	7	**0**
False Incorrect	**3**	7	8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

Total number of benchmarks in which the program does not contain errors

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct				15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	7	8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

ESBMC-GPU achieves the highest "True Correct" results

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
			9	7	**0**
			8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

Total number of benchmarks in which the error in the program was found and an error path was reported

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect				7	**0**
False Incorrect				11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

ESBMC-GPU detects data race, array out of bounds, null pointer, and user-specified assertion

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
[illegible]	[illegible]	[illegible]	8	11	**3**
[illegible]	[illegible]	[illegible]	49	82	104
Time(s)	811	128	147	**12**	158

Total number of benchmarks in which the program had an error but the verifier did not find it

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	7			**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

CIVL did not present any "True Incorrect" result

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
		56	30	15	24
		14	9	7	**0**
False Incorrect	**3**	7	8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

Total number of benchmarks in which an error is reported for a program that fulfills the specification

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect				7	**0**
False Incorrect	**3**	7	8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

ESBMC-GPU and CIVL present the lowest "False Incorrect" results

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
			8	11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

Total number of benchmarks which are not supported by the tool

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect				11	**3**
Not supported	**23**	24	49	82	104
Time(s)	811	128	147	**12**	158

ESBMC-GPU supports the largest number of benchmarks

Experimental Results

Result\Tool	ESBMC-GPU	GKLEE	GPUVERIFY	PUG	CIVL
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	7			
Not supported	**23**	24			
Time(s)	811	128	147	**12**	158

PUG is the fastest verifier, but it does not present the highest covegare

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs
  - First software verifier that uses:
    - SMT-based Context-BMC for verifying CUDA-based programs
    - MPOR, responsible for reducing 80% of the verification time

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs
  - First software verifier that uses:
    - SMT-based Context-BMC for verifying CUDA-based programs
    - MPOR, responsible for reducing 80% of the verification time
- ESBMC-GPU produces 82.5% of successful verification rate compared to 70.8% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs
  - First software verifier that uses:
    - SMT-based Context-BMC for verifying CUDA-based programs
    - MPOR, responsible for reducing 80% of the verification time
- ESBMC-GPU produces 82.5% of successful verification rate compared to 70.8% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL
- Future work
  - Barrier divergence detection

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs
  - First software verifier that uses:
    - SMT-based Context-BMC for verifying CUDA-based programs
    - MPOR, responsible for reducing 80% of the verification time
- ESBMC-GPU produces 82.5% of successful verification rate compared to 70.8% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL
- Future work
  - Barrier divergence detection
  - Support to new memory types (*e.g.,* pinned and unified)

Conclusions

- Proposed a software verifier, which is able to check CUDA-based programs
  - First software verifier that uses:
    - SMT-based Context-BMC for verifying CUDA-based programs
    - MPOR, responsible for reducing 80% of the verification time
- ESBMC-GPU produces 82.5% of successful verification rate compared to 70.8% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL
- Future work
  - Barrier divergence detection
  - Support to new memory types (*e.g.,* pinned and unified)
  - Techniques to reduce interleavings (lazy sequentialization)