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

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

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

- 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)

Exploit SMT-based context-BMC to verify CUDA-based programs
Objectives of this work

- 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 ESBMCQtOM (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\textsuperscript{QtOM} (SPIN’16)

• CUDA is a **proprietary platform**
  – CUDA Programming Guide and IDE Nsight
```c
__global__ void kernel(uint4 *out) {
    uint4 vector = {1,1,1,1};
    out[threadIdx.x] = vector;
}

int main(){
    uint4 *a;
    ...
```
struct __device_builtin__ __builtin_align__(16) _uint4{
  unsigned int x,
  y,
  z,
  w;
};

typedef __device_builtin__
struct _uint4
uint4;

...__global__ void kernel(uint4 *out) {
    uint4 vector = {1,1,1,1};
    out[threadIdx.x] = vector;
}

int main(){
    uint4 *a;
    ...
}
struct __device_builtin__ __builtin_align__(16) _uint4{
    unsigned int x, y, z, w;
};

typedef __device_builtin__ struct _uint4 uint4;

... __global__ void kernel(uint4 *out) {
    uint4 vector = {1,1,1,1};
    out[threadIdx.x] = vector;
}
int main(){
    uint4 *a;
    ...

ESBMC-GPU: Verification Flow

Source Code

Operational Model

ESBMC

Error?

Counterexample

Verification Successful
struct __device_builtin__ __builtin_align__(16) _uint4{
    unsigned int x, y, z, w;
};

typedef __device_builtin__ struct _uint4 uint4;

...__global__ void kernel(uint4 *out) {
    uint4 vector = {1,1,1,1};
    out[threadIdx.x] = vector;
}

int main(){
    uint4 *a;
    ...

ESBMC-GPU: Verification Flow

Source Code

ESBMC

Error?

Yes

Counterexample

State 319 thread 0
<main invocation>
----------------------------------
c::main::$tmp::tmp$2=FALSE
----------------------------------

State 320 file main.cu line 31 thread 0
<main invocation>
----------------------------------
Violated property:
file main.cu line 31 assertion FALSE
----------------------------------

VERIFICATION FAILED

Verification Successful

Symex completed in: 0.029s
size of program expression: 732 assignments
Slicing time: 0.003s
Generated 237 VCC(s), 167 remaining after simplification
No solver specified; defaulting to z3
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.008s
Solving with solver Z3 v4.0
Runtime decision procedure: 0.007s
VERIFICATION SUCCESSFUL
BMC program time: 0.051s
COM Implementation: `cudaMalloc`

```c
#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,  
    cudaMemcpyHostToDevice, sizeof(int), cudaMemcpyHostToDevice);  
    ESBMC_verify_kernel(definitions,1,N,dev_a);  
    cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyDeviceToDevice);  
    assert(a==25);  
    cudaFree(dev_a);  
    return 0;  
}
```
#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;
    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),cudaMemcpyDeviceToHost);
    assert(a==25);
    cudaFree(dev_a);
    return 0;
}
COM Implementation: `cudaMalloc`

```c
# 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**
# 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;
}
# 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

```c
#include <cuda_runtime.h>

__global__ void kernel()
{
    A[tid.x] = tid.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

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

```c
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

<table>
<thead>
<tr>
<th>Function conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMalloc(&amp;dev_a,size)</td>
</tr>
<tr>
<td>assert(size&gt;0);</td>
</tr>
<tr>
<td>*dev_a=malloc(size);</td>
</tr>
<tr>
<td>if(*dev_a==NULL)</td>
</tr>
<tr>
<td>exit(1);</td>
</tr>
</tbody>
</table>

- 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**

```c
__global__ void kernel(){
    A[tid.x]=tid.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

  ```c
cudaMalloc(&dev_a,size)

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

**ESBMC_verify_kernel**

```c
(kernel<M,N,dev_a)

gridDim = dim3(M);
blockDim = dim3(N);
```

**dim3 conversion**

```c
struct dim3;
gridDim.x=M; blockDim.x=N;
gridDim.y=1; blockDim.y=1;
gridDim.z=1; blockDim.z=1;
```
Modeling Kernels with Pthreads in COM

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

CUDA program

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

```c
int main(){
    int *a; int *dev_a;
    cudaMalloc(&dev_a,a,size);
    cudaMemcp(dy_a,a,htd);
    ESBMC_verify_kernel(kernel,M,N,dev_a);
    cudaMemcp(a,dev_a,dth);
    cudaFree(dev_a);
    free(a);
}
```

COM

Function conversion

```c
cudaMalloc(&dev_a,size)
```

```c
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

CUDA program

```c
_global_ void kernel(){
    A[tid.x]=tid.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);
    ... cudaFree(dev_a);
    free(a);
}
```

COM

Function conversion

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

ESBMC_verify_kernel

```c
ESBMC_verify_kernel_wta(gridDim.x*gridDim.y*gridDim.z,blockDim.x*blockDim.y*blockDim.z,arg1,arg2,arg3)
```

ESBMC_verify_kernel_wta

```c
while(i<GPU_threads){
    pthread_create(&threads_id,
                  NULL, kernel, NULL);
    i++;
}
```
Modeling Kernels with Pthreads in COM

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

CUDA program

```c
_global_ void kernel()
{
    A[tid.x]=tid.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);
    ... 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);
```

```
kernell<<M,N>>>
```

| `gridDim = dim3(M);` |
| `blockDim = dim3(N);` |

dim3 conversion

```
struct dim3;
gridDim.x=M; blockDim.x=N;
gridDim.y=1; blockDim.y=1;
gridDim.z=1; blockDim.z=1;
```

ESBMC

```
while(i<GPU_threads){
    pthread_create(&threads_id, NULL, kernel, NULL);
    i++;
}
```

ESBMC_verify_kernel_wta

```
ESBMC_verify_kernel_wta(
    gridDim.x*gridDim.y*gridDim.z, blockDim.x*blockDim.y,blockDim.z, arg1,arg2,arg3)
```

ESBMC_verify_kernel

```
(kernel,M,N,dev_a)
```

Calls the auxiliary function

```
ESBMC_verify_kernel(kernel,M,N,dev_a);
```

- 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 $γ(s_{i−1}, s_i)$ 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

1. function MPOR (ν, π)  \( π = \{ν_0,...,ν_k\} \)
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 \( γ(s_{i-1},s_i) \) 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

\( ν = (A_i,C_i,s_i) \)

- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

\[
\begin{align*}
\text{kernel1(int *a) \{} \\
\quad a[\text{threadIdx.x}] &= \text{threadIdx.x} ; \\
\} \\
v_0: t_0, 0, a[0] = 0, a[1] = 0
\end{align*}
\]
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

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

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

\( v = (A_i, C_i, s_i) \)

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

\[ v_1 : t_1, 1, a[0] = 0, a[1] = 0 \]
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

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

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

\( v_0: t_0, 0, a[0] = 0, a[1] = 0 \)

\( v_1: t_1, 1, a[0] = 0, a[1] = 0 \)

\( v_2: t_2, 2, a[0] = 0, a[1] = 1 \)
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

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

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

= (A_i, C_i, s_i)

\( A_i \): active thread
\( C_i \): context switch
\( s_i \): current state

\( v_0; t_0, 0, a[0] = 0, a[1] = 0 \)

\( v_1; t_1, 1, a[0] = 0, a[1] = 0 \)

\( v_2; t_2, 2, a[0] = 0, a[1] = 1 \)
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

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

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

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

\( \nu = (A_i, C_i, s_i) \)

- \( t_1 \rightarrow t_2 \)
- \( s_2: a[0] = 0 \)
- \( a[1] = 1 \)

\( v_0: t_0, 0, a[0] = 0 \), \( a[1] = 0 \)

\( v_1: t_1, 1, a[0] = 0 \), \( a[1] = 0 \)

\( v_2: t_2, 2, a[0] = 0 \), \( a[1] = 1 \)
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

1. **function** MPOR (ν, π) \( π = \{v_0, ..., v_k\} \)
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 \( γ(s_{i-1}, s_i) \) is independent on \( π \); otherwise, go to step 6
5. Return “independent” on \( π \) and terminates
6. Return “dependent” on \( π \) and terminates

**kernel1(int *a){**

\[
\begin{align*}
    &a[\text{threadIdx.x}] = \text{threadIdx.x} ; \\
\end{align*}
\]

}  

**Dependent**

\( t_1 \to t_2 \)

\( s_2: \ a[0] = 0 \)

\( a[1] = 1 \)

\( v_0: t_0, 0, a[0] = 0, a[1] = 0 \)

\( v_1: t_1, 1, a[0] = 0, a[1] = 0 \)

\( v_2: t_2, 2, a[0] = 0, a[1] = 1 \)

\( \nu = (A_i, C_i, s_i) \)

- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state
MPOR Applied to CUDA-based Programs

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

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

\( v = (A_i, C_i, s_i) \)

- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state

**States:**
- \( v_0: t_0, 0, a[0] = 0, a[1] = 0 \)
- \( v_1: t_1, 1, a[0] = 0, a[1] = 0 \)
- \( v_2: t_2, 2, a[0] = 0, a[1] = 1 \)
- \( v_3: t_2, 1, a[0] = 0, a[1] = 1 \)
MPOR Applied to CUDA-based Programs

**MPOR algorithm in the ESBMC-GPU**

1. **function** MPOR ($\nu$, $\pi$)  
   $$\pi = \{\nu_0, ..., \nu_k\}$$
2. Check whether $s_i$ exists in $\pi$; otherwise, go to step 4
3. Check whether $A_i$ produces a new state in $\pi$; otherwise, go to step 5
4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on $\pi$; otherwise, go to step 6
5. Return “independent” on $\pi$ and terminates
6. Return “dependent” on $\pi$ and terminates
7. **end function**

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

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

Dependent

$t_1 \rightarrow t_2$

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

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

$v_1$: $t_1$, 1, $a[0] = 0$, $a[1] = 0$

$v_2$: $t_2$, 2, $a[0] = 0$, $a[1] = 1$

$v_3$: $t_2$, 1, $a[0] = 0$, $a[1] = 1$

$v_4$: $t_2$, 1, $a[0] = 0$, $a[1] = 1$
MPOR algorithm in the ESBMC-GPU

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

\[
v = (A_i, C_i, s_i)
\]

- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state

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

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

\[
v_0: t_0, 0, a[0] = 0, a[1] = 0
\]
\[
v_1: t_1, 1, a[0] = 0, a[1] = 0
\]
\[
v_2: t_2, 2, a[0] = 0, a[1] = 1
\]
\[
v_3: t_1, 1, a[0] = 0, a[1] = 1
\]
\[
v_4: t_1, 2, a[0] = 0, a[1] = 1
\]
### MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

\[
\begin{align*}
\text{kernel1}(\text{int} \ast a)\{ \\
& a[\text{threadIdx.x}] = \text{threadIdx.x} ; \\
\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>(v_0:) (t_0, 0, a[0] = 0, a[1] = 0)</th>
<th>(v_1:) (t_1, 1, a[0] = 0, a[1] = 0)</th>
<th>(v_3:) (t_2, 1, a[0] = 0, a[1] = 1)</th>
<th>(v_4:) (t_1, 2, a[0] = 0, a[1] = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_2:) (t_2, 2, a[0] = 0, a[1] = 1)</td>
<td>(v_0:) (t_0, 0, a[0] = 0, a[1] = 0)</td>
<td>(v_4:) (t_1, 2, a[0] = 0, a[1] = 1)</td>
<td>(v_4:) (t_1, 2, a[0] = 0, a[1] = 1)</td>
</tr>
</tbody>
</table>

- **Dependent**
  - \(t_1 \rightarrow t_2\)
  - \(t_2 \rightarrow t_1\)
- \(s_2:\) \(a[0] = 0\)
  - \(a[1] = 1\)
- \(s_2:\) \(a[0] = 0\)
  - \(a[1] = 1\)
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

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

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

\[
\text{kernel1(int*a)}
\begin{align*}
\text{a[threadIdx.x]} &= \text{threadIdx.x} ; \\
\end{align*}
\]

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

\[ v_0: t_0, 0, a[0] = 0, a[1] = 0 \]
\[ v_1: t_1, 1, a[0] = 0, a[1] = 0 \]
\[ v_2: t_2, 2, a[0] = 0, a[1] = 1 \]
\[ v_3: t_1, 1, a[0] = 0, a[1] = 1 \]
\[ v_4: t_1, 2, a[0] = 0, a[1] = 1 \]

\( t_1 \rightarrow t_2 \) 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
  1. function MPOR (ν, π)  \( \pi = \{v_0, ..., v_k\} \)
  2. Check whether \( s_i \) exists in \( \pi \); otherwise, go to step 4
  3. Check whether \( A_i \) produces a new state in \( \pi \); otherwise, go to step 5
  4. Analyze whether \( \gamma(s_{i-1}, s_i) \) is independent on \( \pi \); otherwise, go to step 6
  5. Return “independent” on \( \pi \) and terminates
  6. Return “dependent” on \( \pi \) and terminates
  7. end function

\[
\nu = (A_i, C_i, s_i)
\]

\( A_i \): 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;

\[
\nu_0: t_0, 0, a[0] = 0, a[1] = 0
\]
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

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

$v = (A_i, C_i, s_i)$

$A_i$: active thread
$C_i$: context switch
$s_i$: current state

$v_0$: $t_0$, $a[0] = 0$, $a[1] = 0$
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

1. function MPOR (ν, π) π = {ν₀,...,νₖ}
2. Check whether sᵢ exists in π; otherwise, go to step 4
3. Check whether Aᵢ produces a new state in π; otherwise, go to step 5
4. Analyze whether γ(sᵢ₋₁, sᵢ) 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;

ν₀: t₀, 0, a[0] = 0 , a[1] = 0
ν₁: t₁, 1, a[0] = 0 , a[1] = 0
ν = (Aᵢ, Cᵢ, sᵢ)
Aᵢ: active thread
Cᵢ: context switch
sᵢ: current state
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

1. **function** MPOR (ν, π) \( \pi = \{\nu_0, ..., \nu_k\} \)
2. Check whether \( s_i \) exists in \( \pi \); otherwise, go to step 4
3. Check whether \( A_i \) produces a new state in \( \pi \); otherwise, go to step 5
4. Analyze whether \( \gamma(s_{i-1}, s_i) \) is independent on \( \pi \); otherwise, go to step 6
5. Return “independent” on \( \pi \) and terminates
6. Return “dependent” on \( \pi \) and terminates
7. **end function**

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

\( \nu = (A_i, C_i, s_i) \)

- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

1. function MPOR (ν, π)  \[ π = \{v_0, \ldots, v_k\} \]
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 \( γ(s_{i-1}, s_i) \) 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;

\[ \nu = (A_i, C_i, s_i) \]
\( A_i \): active thread
\( C_i \): context switch
\( s_i \): current state

\[ ν_0: t_0, 0, a[0] = 0, a[1] = 0 \]
\[ ν_1: t_1, 1, a[0] = 0, a[1] = 0 \]
\[ ν_2: t_2, 2, a[0] = 0, a[1] = 1 \]
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

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

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

\( \nu = (A_i, C_i, s_i) \)
\( A_i \): active thread
\( C_i \): context switch
\( s_i \): current state
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

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

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

\( \nu_0: t_0, 0, a[0] = 0, a[1] = 0 \)
\( \nu_1: t_1, 1, a[0] = 0, a[1] = 0 \)
\( \nu_2: t_2, 2, a[0] = 0, a[1] = 1 \)
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

1. function MPOR \( (v, \pi) \)
   \( \pi = \{v_0, ..., v_k\} \)
2. Check whether \( s_i \) exists in \( \pi \); otherwise, go to step 4
3. Check whether \( A_i \) produces a new state in \( \pi \); otherwise, go to step 5
4. Analyze whether \( \gamma(s_{i-1}, s_i) \) is independent on \( \pi \); otherwise, go to step 6
5. Return “independent” on \( \pi \) and terminates
6. Return “dependent” on \( \pi \) 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
\( t_1 \rightarrow t_2 \)
\( s_2: a[0] = 0 \)
\( a[1] = 1 \)

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

\( v_0: t_0, 0, a[0] = 0, a[1] = 0 \)
\( v_1: t_1, 1, a[0] = 0, a[1] = 0 \)
\( v_2: t_2, 2, a[0] = 0, a[1] = 1 \)
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

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

```
kernell (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

\[v = (A_i, C_i, s_i)\]

- \(A_i\): active thread
- \(C_i\): context switch
- \(s_i\): current state
MPOR Applied to CUDA-based Programs

- **MPOR algorithm in the ESBMC-GPU**

1. function MPOR \((v, \pi)\) \(\pi = \{\nu_0, ..., \nu_k\}\)
2. Check whether \(s_i\) exists in \(\pi\); otherwise, go to step 4
3. Check whether \(A_i\) produces a new state in \(\pi\); otherwise, go to step 5
4. Analyze whether \(\gamma(s_{i-1}, s_i)\) is independent on \(\pi\); otherwise, go to step 6
5. Return “independent” on \(\pi\) and terminates
6. Return “dependent” on \(\pi\) and terminates
7. end function

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

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

**kernel (int *a)**

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

**Dependent**

- \( t_1 \rightarrow t_2 \)
- \( s_2: a[0] = 0 \)
- \( a[1] = 1 \)

**Example State Transition**

- \( v_0: t_0, 0, a[0] = 0, a[1] = 0 \)
- \( v_1: t_1, 1, a[0] = 0, a[1] = 0 \)
- \( v_2: t_2, 2, a[0] = 0, a[1] = 1 \)
- \( v_3: t_1, 1, a[0] = 0, a[1] = 1 \)

**Variables**

- \( v = (A_i, C_i, s_i) \)
- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU

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

2. Check whether $s_i$ exists in $\pi$; otherwise, go to step 4

3. Check whether $A_i$ produces a new state in $\pi$; otherwise, go to step 5

4. Analyze whether $\gamma(s_{i-1}, s_i)$ is independent on $\pi$; otherwise, go to step 6

5. Return “independent” on $\pi$ and terminates

6. Return “dependent” on $\pi$ and terminates

7. **end function**

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

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

$\nu = (v_0, v_1, v_2, v_3, v_4)$

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 (ν, π) π = {ν₀,...,νₖ}
2. ➔ Check whether sᵢ exists in π; otherwise, go to step 4
3. Check whether Aᵢ produces a new state in π; otherwise, go to step 5
4. Analyze whether γ(sᵢ₋₁, sᵢ) 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

v₀: t₀, 0, a[0] = 0, a[1] = 0
v₁: t₁, 1, a[0] = 0, a[1] = 0
v₂: t₂, 2, a[0] = 0, a[1] = 1
v₃: t₁, 1, a[0] = 0, a[1] = 1
v₄: t₁, 2, a[2] = 0, a[1] = 1

v = (Aᵢ, Cᵢ, sᵢ)
Aᵢ: active thread
Cᵢ: context switch
sᵢ: current state
MPOR Applied to CUDA-based Programs

• MPOR algorithm in the ESBMC-GPU
1. function MPOR (ν, π) \( \pi = \{\nu_0, \ldots, \nu_k\} \)
2. Check whether \( s_i \) exists in \( \pi \); otherwise, go to step 4
3. Check whether \( A_i \) produces a new state in \( \pi \); otherwise, go to step 5
4. \( \rightarrow \) Analyze whether \( \gamma(s_{i-1}, s_i) \) is independent on \( \pi \); otherwise, go to step 6
5. Return “independent” on \( \pi \) and terminates
6. Return “dependent” on \( \pi \) and terminates
7. end function

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

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

<table>
<thead>
<tr>
<th>Dependent</th>
<th>t_1 (\rightarrow) t_2</th>
<th>t_2 (\rightarrow) t_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_2: a[0] = 0</td>
<td>s_3: a[2] = 0</td>
<td></td>
</tr>
<tr>
<td>a[1] = 1</td>
<td>a[1] = 1</td>
<td></td>
</tr>
</tbody>
</table>

\(v_0: t_0, 0, a[0] = 0, a[1] = 0\)
\(v_1: t_1, 1, a[0] = 0, a[1] = 0\)
\(v_2: t_2, 2, a[0] = 0, a[1] = 1\)
\(v_3: t_1, 1, a[0] = 0, a[1] = 1\)
\(v_4: t_1, 2, a[2] = 0, a[1] = 1\)
MPOR Applied to CUDA-based Programs

- MPOR algorithm in the ESBMC-GPU

1. **function** MPOR \( (\nu, \pi) \)  
   \( \pi = \{\nu_0, ..., \nu_k\} \)
2. Check whether \( s_i \) exists in \( \pi \); otherwise, go to step 4
3. Check whether \( A_i \) produces a new state in \( \pi \); otherwise, go to step 5
4. Analyze whether \( \gamma(s_{i-1}, s_i) \) is independent on \( \pi \); otherwise, go to step 6
5. Return “independent” on \( \pi \) and terminates
6. **Return “dependent”** on \( \pi \) and terminates
7. **end function**

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

\( \nu = (A_i, C_i, s_i) \)
- \( A_i \): active thread
- \( C_i \): context switch
- \( s_i \): current state

Dependent
- \( t_1 \rightarrow t_2 \)  
  \( t_2 \rightarrow t_1 \)
- \( s_2: a[0] = 0 \)  
  \( s_3: a[2] = 0 \)
- \( a[1] = 1 \)  
  \( a[1] = 1 \)

\( t_1 \rightarrow t_2 \) 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

Instruction Cache
- Scheduler
- Scheduler

Dispatch
- Dispatch

Register File
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core

Fermi - Stream Multiprocessor
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

Warp
(32 threads)
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

Block of threads
(64 threads)

Warp
(32 threads)
Two-threads Analysis in Fermi

Block of threads (64 threads)

Warp (32 threads)

One thread group is processed by a half warp in the SM
Two-threads Analysis in Fermi

**Fermi - Stream Multiprocessor**

- **Instruction Cache**
- **Scheduler**
- **Dispatch**
- **Register File**

Block of threads (64 threads)

- **Warp** (32 threads)
- half warp (16 threads)
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

Block of threads (64 threads)

half warp 16 threads

Warp (32 threads)

Memory
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

Block of threads
(64 threads)

half warp
16 threads

Warp
(32 threads)

there is no data race to access different memory positions
Two-threads Analysis in Fermi

Fermi - Stream Multiprocessor

- Instruction Cache
- Scheduler
- Dispatch
- Register File

- Block of threads (64 threads)
  - half warp (16 threads)
- Warp (32 threads)

Memory

access to the same memory position leads to data race
Two-threads Analysis in Fermi

If the error is detected in a half warp threads, it also shows up for two-threads.
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?
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

<table>
<thead>
<tr>
<th>Result \ Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>False Incorrect</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>30</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>False Incorrect</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>False Incorrect</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

ESBMC-GPU achieves the highest “True Correct” results
# Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

Total number of benchmarks in which the error in the program was found and an error path was reported.
## Experimental Results

<table>
<thead>
<tr>
<th>Result/Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>True Incorrect</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>False Incorrect</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

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

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

CIVL did not present any “True Incorrect” result.
# Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Incorrect</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td><strong>12</strong></td>
<td>158</td>
</tr>
</tbody>
</table>

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

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

ESBMC-GPU and CIVL present the lowest “False Incorrect” results.
## Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td>49</td>
<td>82</td>
<td>104</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

**Total number of benchmarks which are not supported by the tool**

- ESBMC-GPU: 23
- GKLEE: 24
- GPUVERIFY: 49
- PUG: 82
- CIVL: 104
# Experimental Results

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

ESBMC-GPU supports the largest number of benchmarks.
# Experimental Results

<table>
<thead>
<tr>
<th>Result\Tool</th>
<th>ESBMC-GPU</th>
<th>GKLEE</th>
<th>GPUVERIFY</th>
<th>PUG</th>
<th>CIVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Correct</td>
<td>60</td>
<td>53</td>
<td>58</td>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>False Correct</td>
<td>67</td>
<td>56</td>
<td>30</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>True Incorrect</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>False Incorrect</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not supported</td>
<td>23</td>
<td>24</td>
<td></td>
<td>82</td>
<td>164</td>
</tr>
<tr>
<td>Time(s)</td>
<td>811</td>
<td>128</td>
<td>147</td>
<td>12</td>
<td>158</td>
</tr>
</tbody>
</table>

PUG is the fastest verifier, but it does not present the highest coverage.
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)