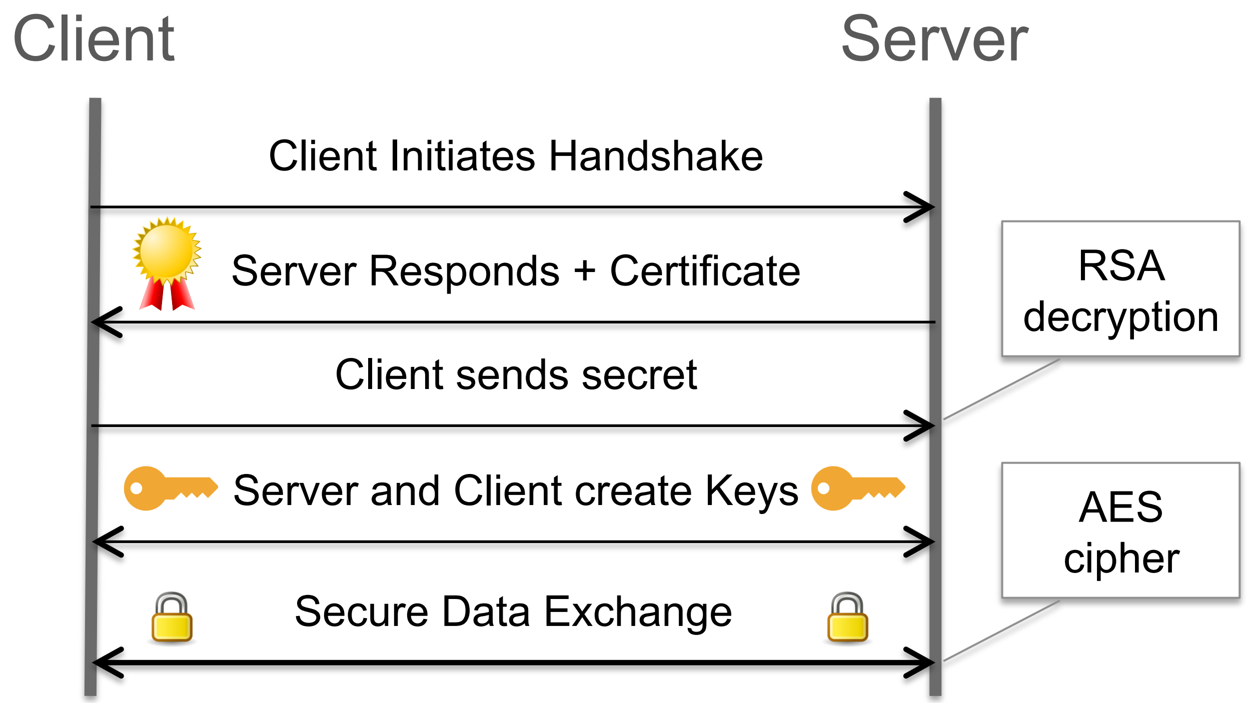
PixelVault: Using GPUs for Securing Cryptographic Operations

Giorgos Vasiliadis **gvasil@ics.forth.gr**
Elias Athanasopoulos elathan@ics.forth.gr
Michalis Polychronakis mikepo@cs.columbia.edu
Sotiris Ioannidis sotiris@ics.forth.gr

How SSL/TLS works

- Secure Sockets Layer (SSL/TLS) is a de-facto standard for secure communication
  - Authentication, confidentiality, integrity

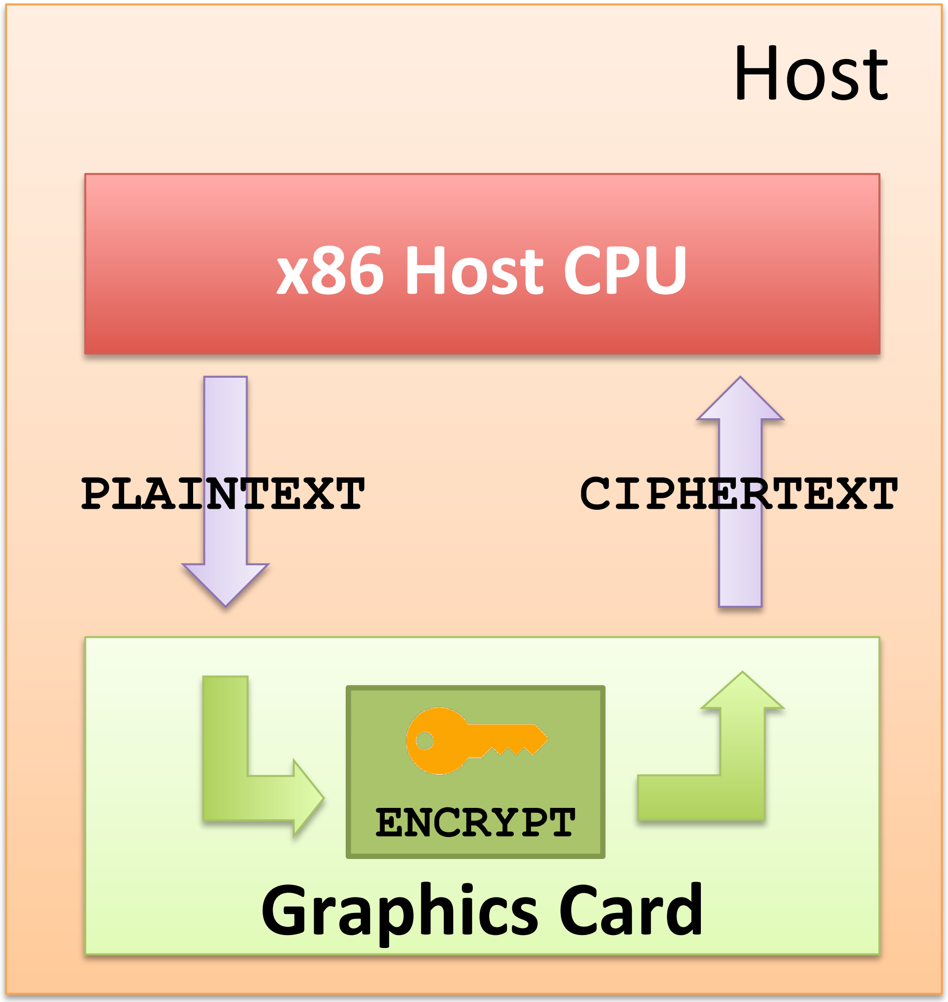


Client
Server
Client Initiates Handshake
Server Responds + Certificate
Client sends secret
Server and Client create Keys
Secure Data Exchange
RSA decryption
AES cipher

Motivation

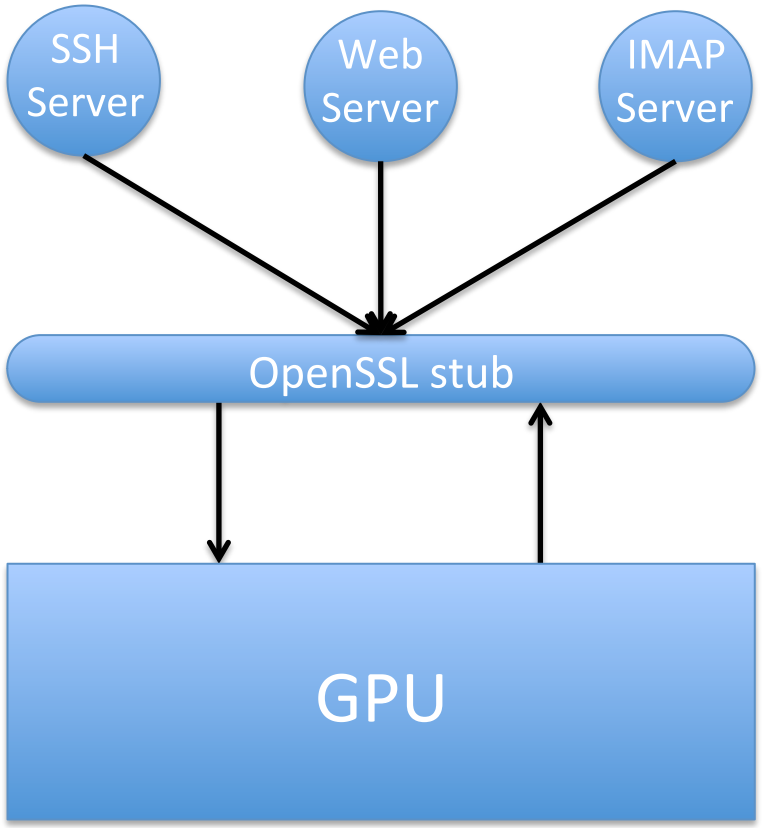
- Secret keys may remain unencrypted in CPU Registers, RAM, etc.
  - Memory attacks
  - DMA/Firewire attacks
  - Heartbleed attack
  - …

PixelVault Overview



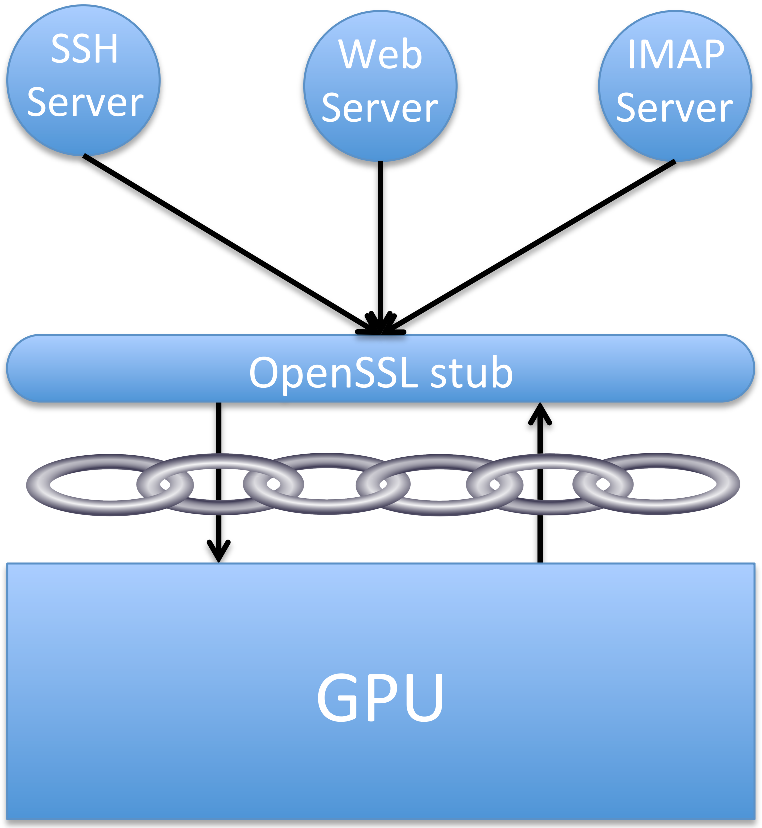
- Runs encryption securely outside CPU/RAM
- Only on-chip memory of GPU is used as storage
- Secret keys are never observed from host

Cryptographic Processing with GPUs



- GPU-accelerated SSL
  - [CryptoGraphics, CT-RSA'05]
  - [Harrison et al., Sec'08]
  - [SSLShader, NSDI'11]
  - …
- High-performance
- Cost-effective

Cryptographic Processing with GPUs



- GPU-accelerated SSL
  - [CryptoGraphics, CT-RSA'05]
  - [Harrison et al., Sec'08]
  - [SSLShader, NSDI'11]
  - …
- High-performance
- Cost-effective

Can we also make it secure?

Implementation Challenges

- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?

Implementation Challenges

- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?

GPU as a coprocessor

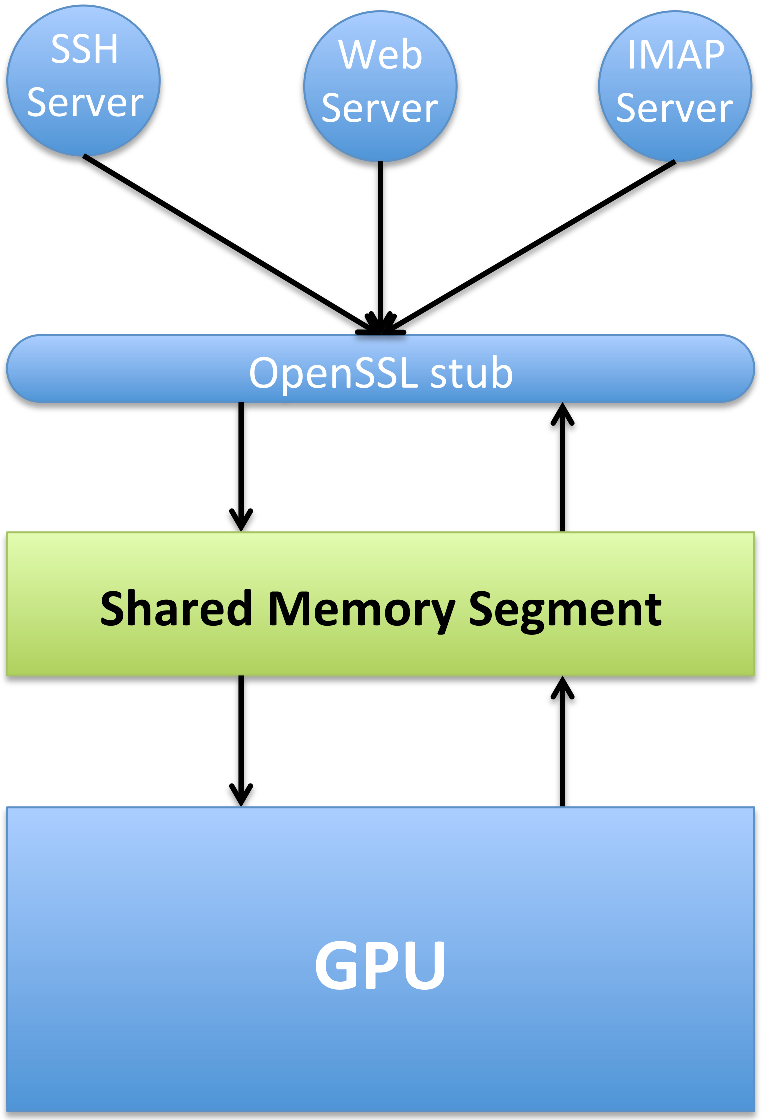
- Typically handled by the host
  - Load parameters, launch GPU kernel, transfer data, etc.
- Not secure for our purposes
  - Crypto keys have to be transferred every time

Autonomous GPU execution

- Force GPU kernel to run indefinitely
  - i.e., using an infinite **`while`** loop

- Cannot rely on the typical parameter-passing execution of GPU kernels
  - Instead, we allocate a memory segment that is shared between CPU/GPU

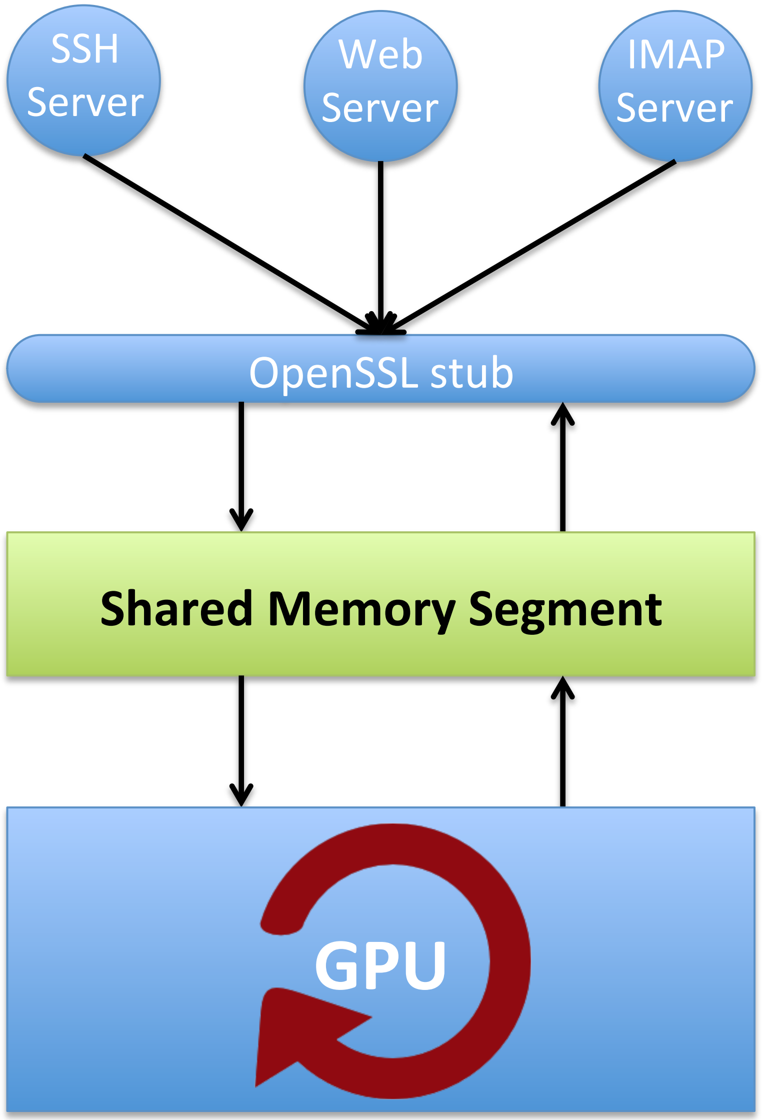
Shared Memory between CPU/GPU



- *Page-locked* memory
  - Accessed by the GPU directly, via DMA
  - Cannot be swapped to disk

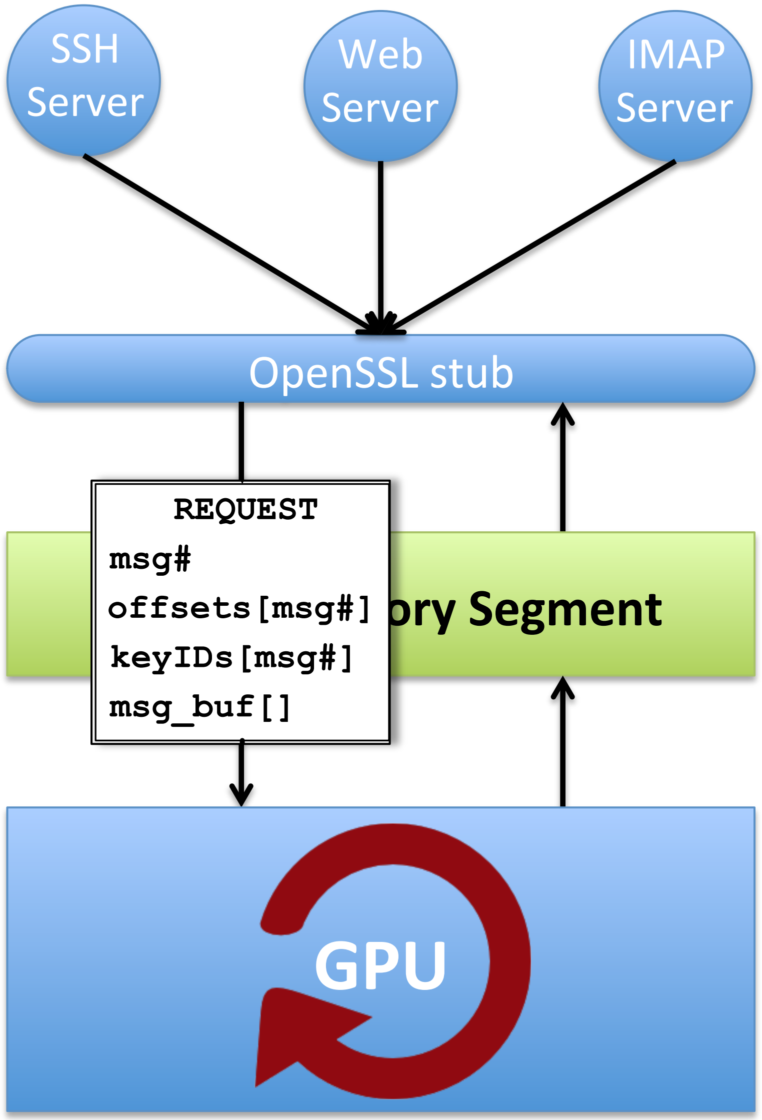
- Processing requests are issued through this shared memory space

Shared Memory between CPU/GPU



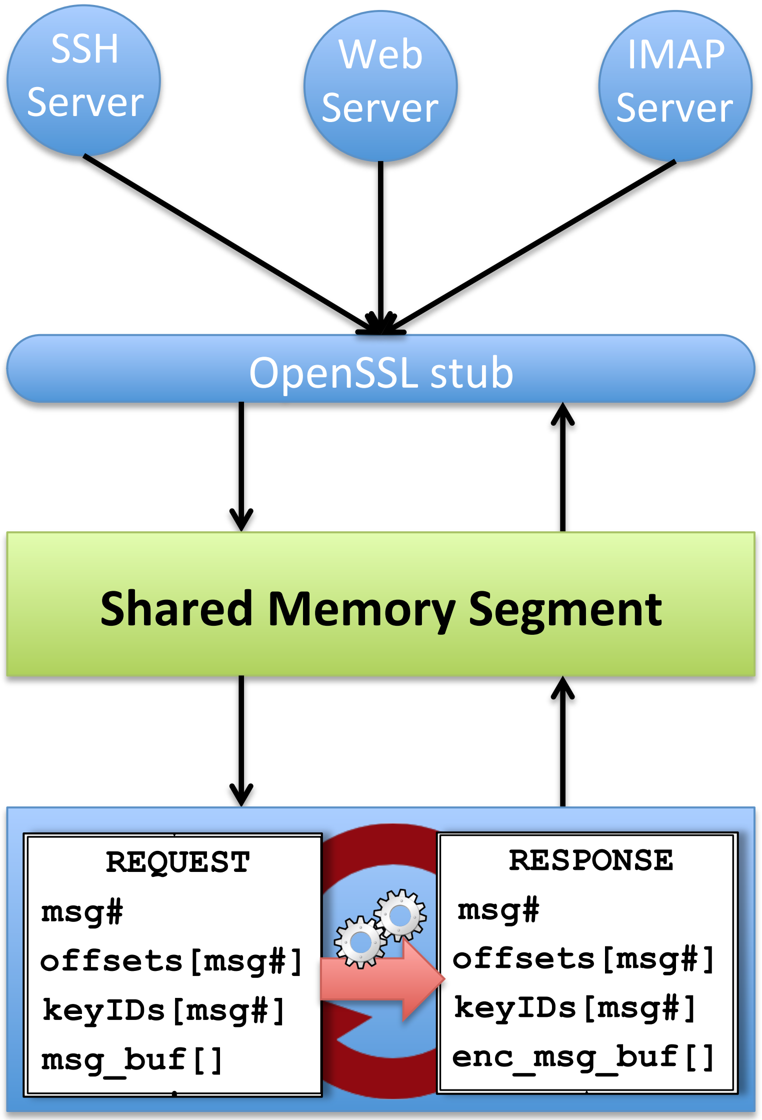
- GPU continuously monitors the shared space for new requests

Shared Memory between CPU/GPU



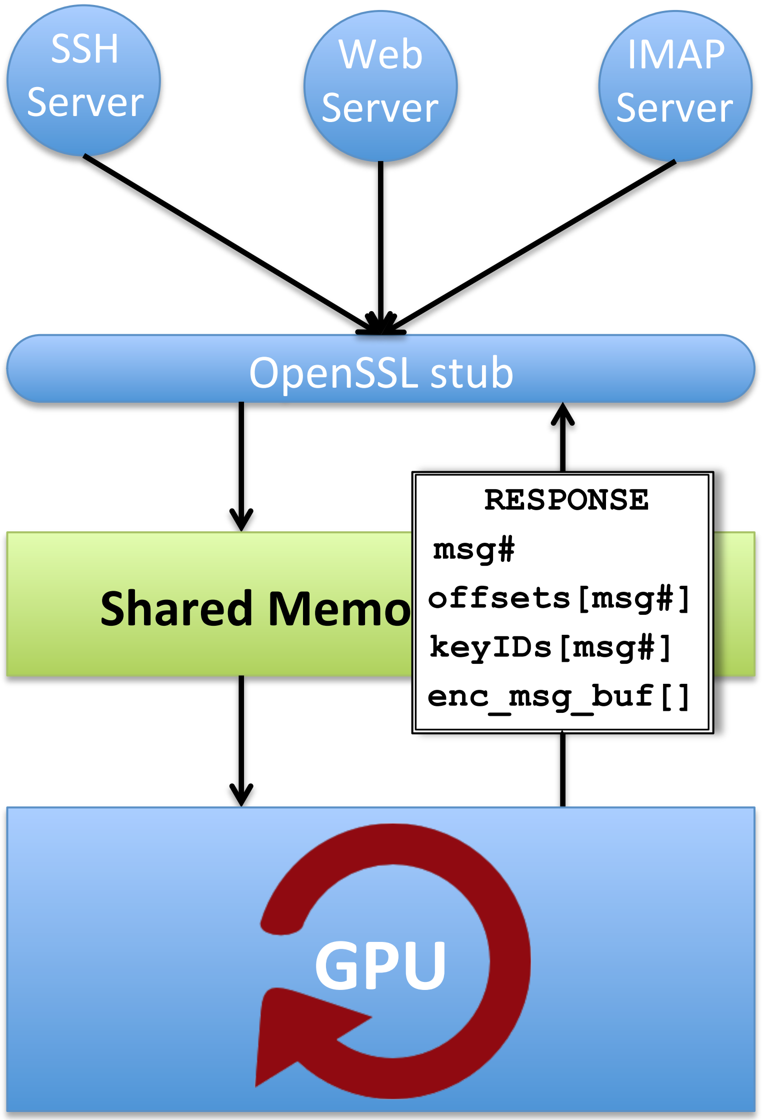
- When a new request is available, it is transferred to the memory space of the GPU

Shared Memory between CPU/GPU



- The request is processed by the GPU

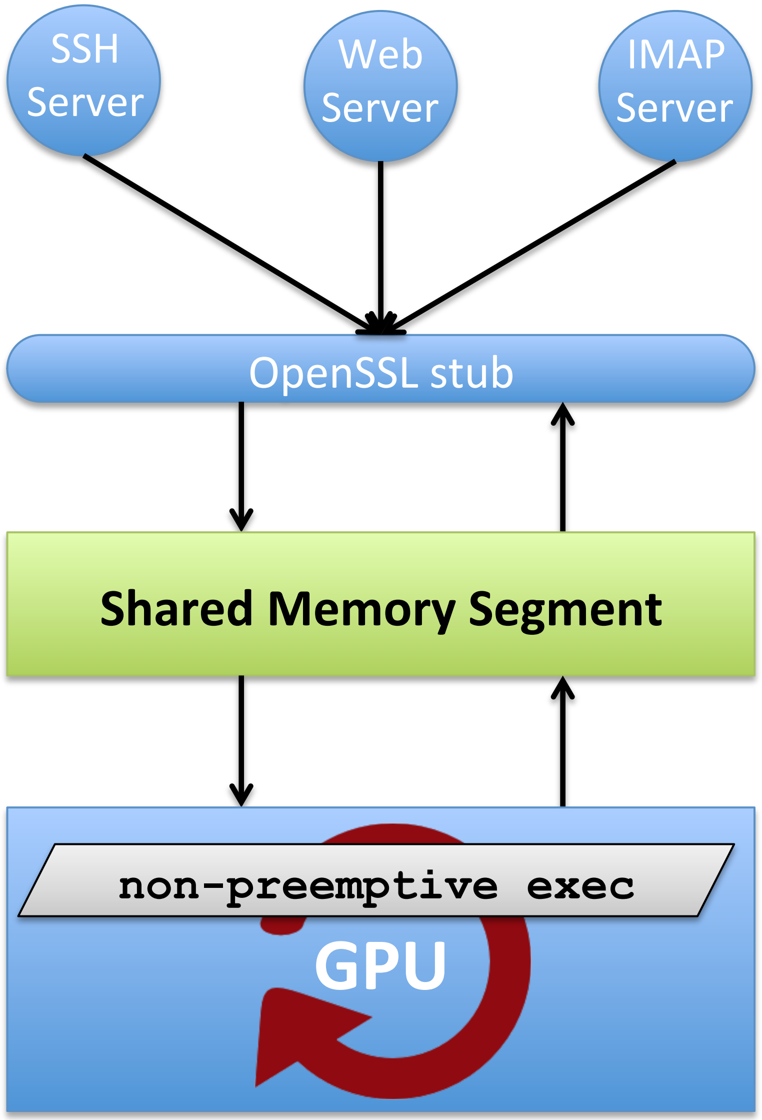
Shared Memory between CPU/GPU



```
RESPONSE
msg#
offsets[msg#]
keyIDs[msg#]
enc_msg_buf[]
```

- When processing is finished, the host is notified by setting the response parameter fields accordingly

Autonomous GPU execution

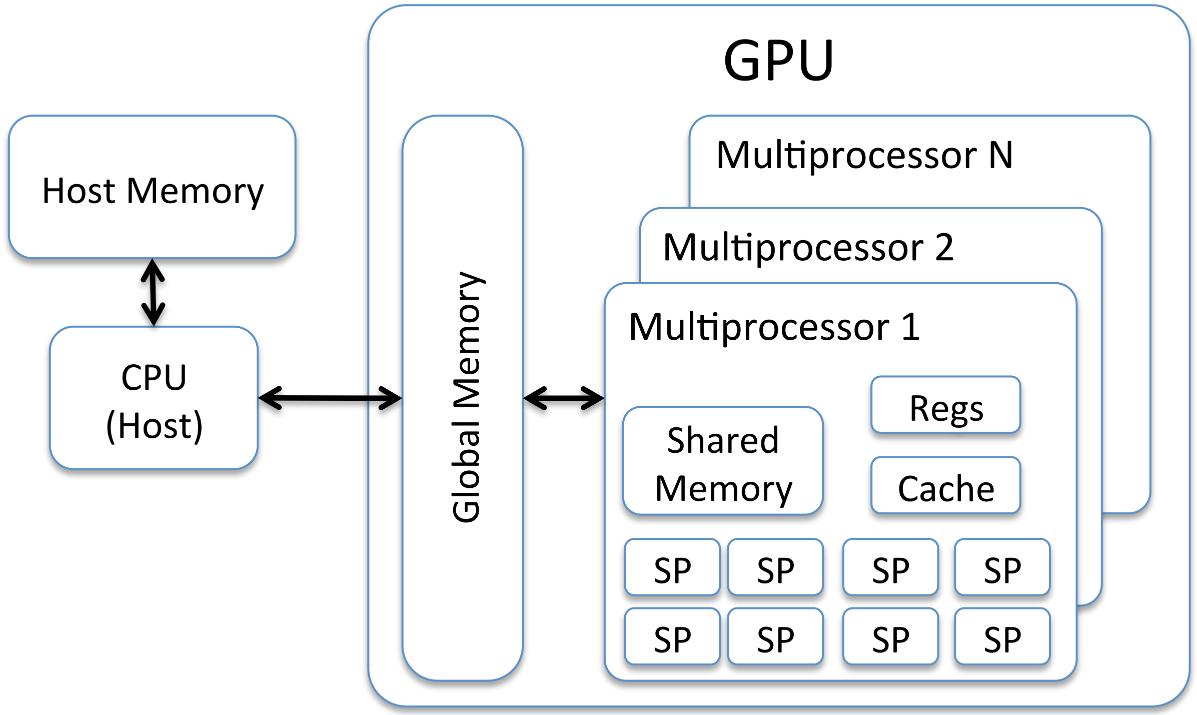


- Non-preemptive execution
- Only the output block is being written back to host memory

Implementation Challenges

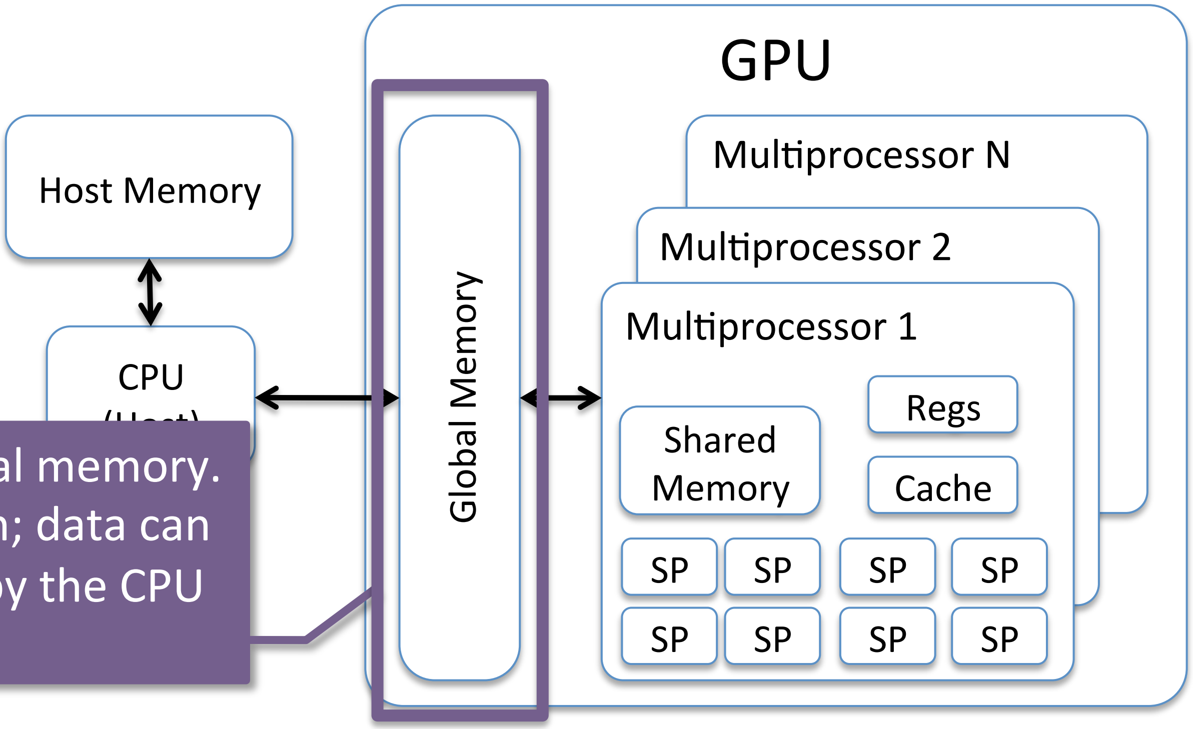
- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?

Who holds the keys?



- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

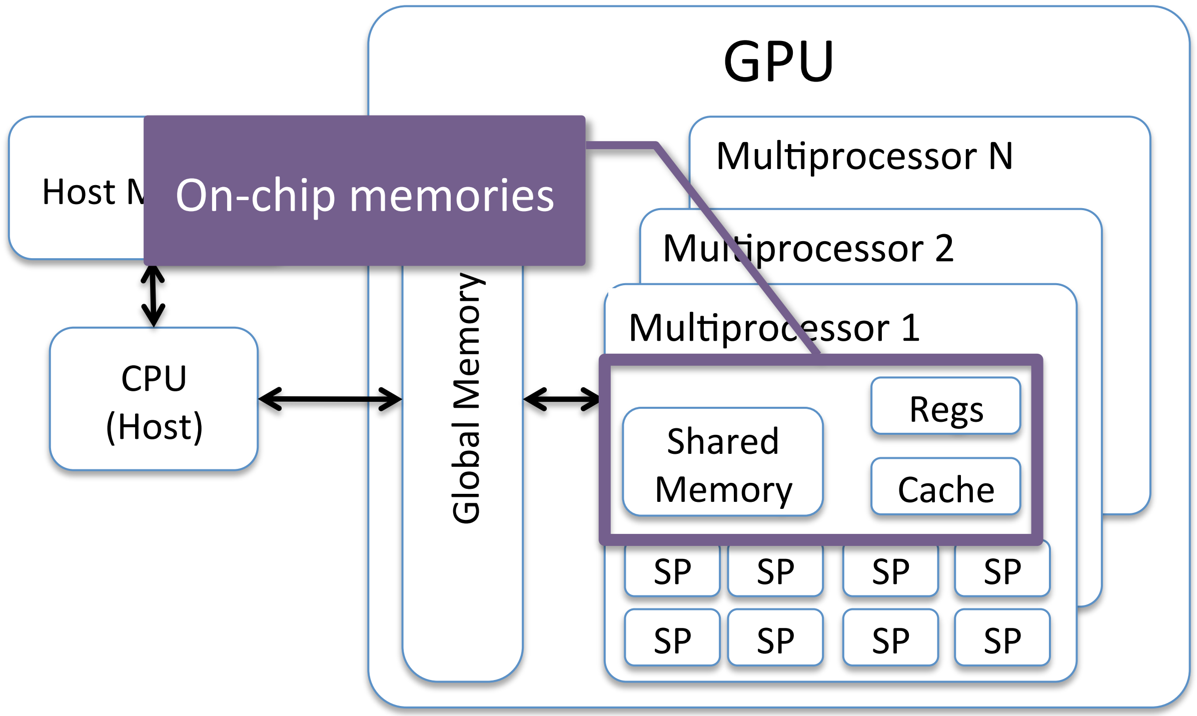
Who holds the keys?



Off-chip global memory. No protection; data can be acquired by the CPU directly.

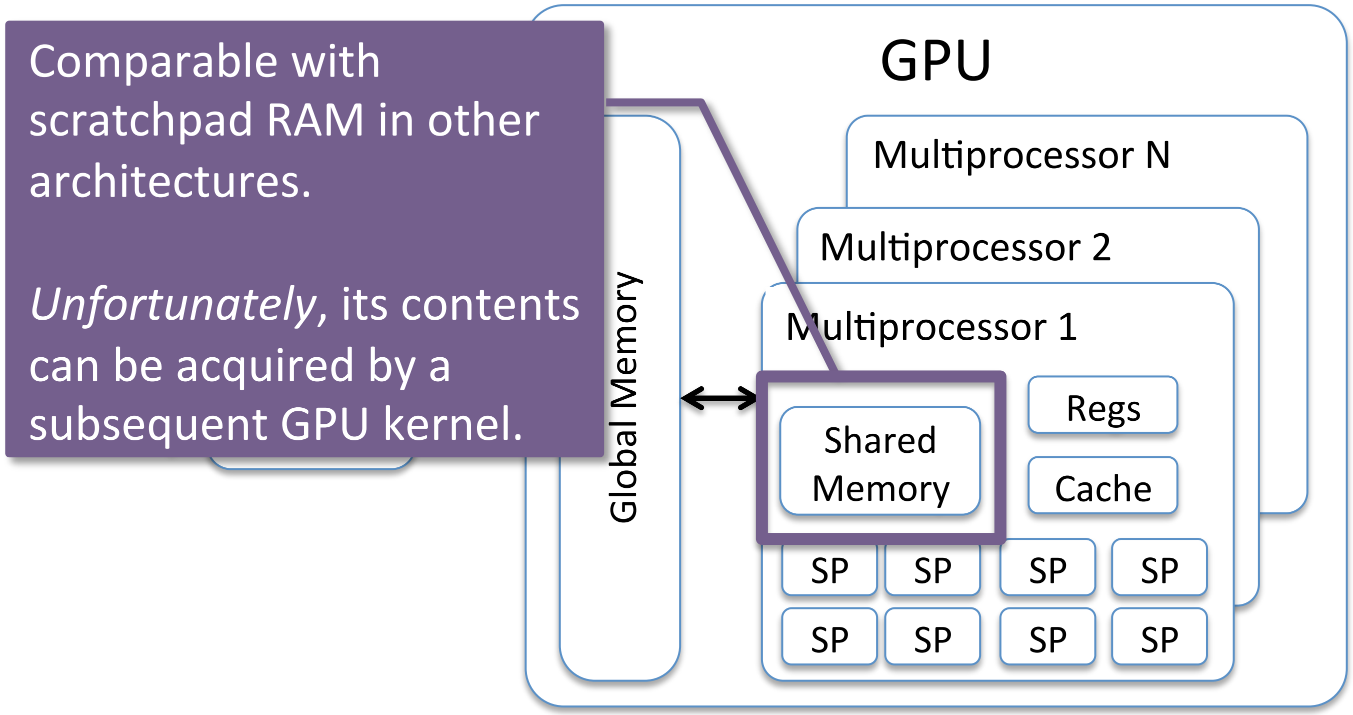
- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

Who holds the keys?



- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

Who holds the keys?

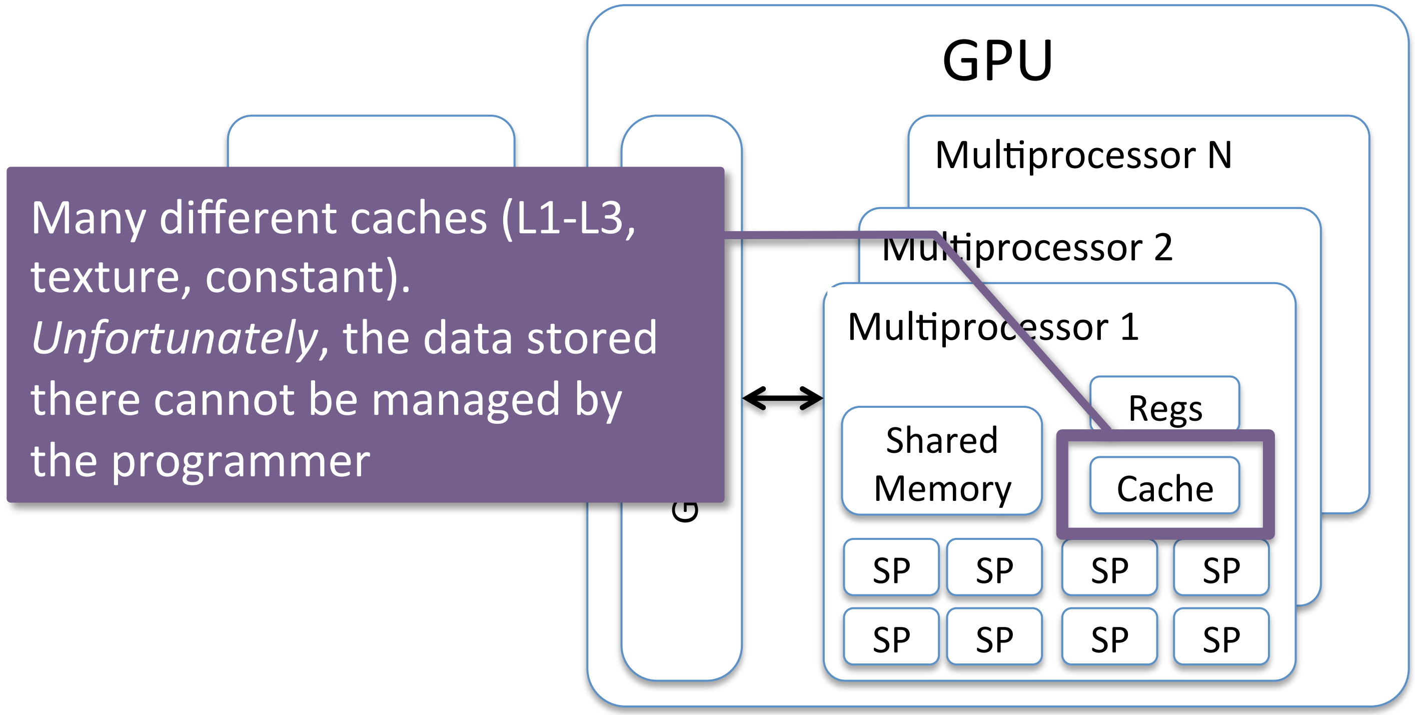


Comparable with scratchpad RAM in other architectures.

Unfortunately, its contents can be acquired by a subsequent GPU kernel.

- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

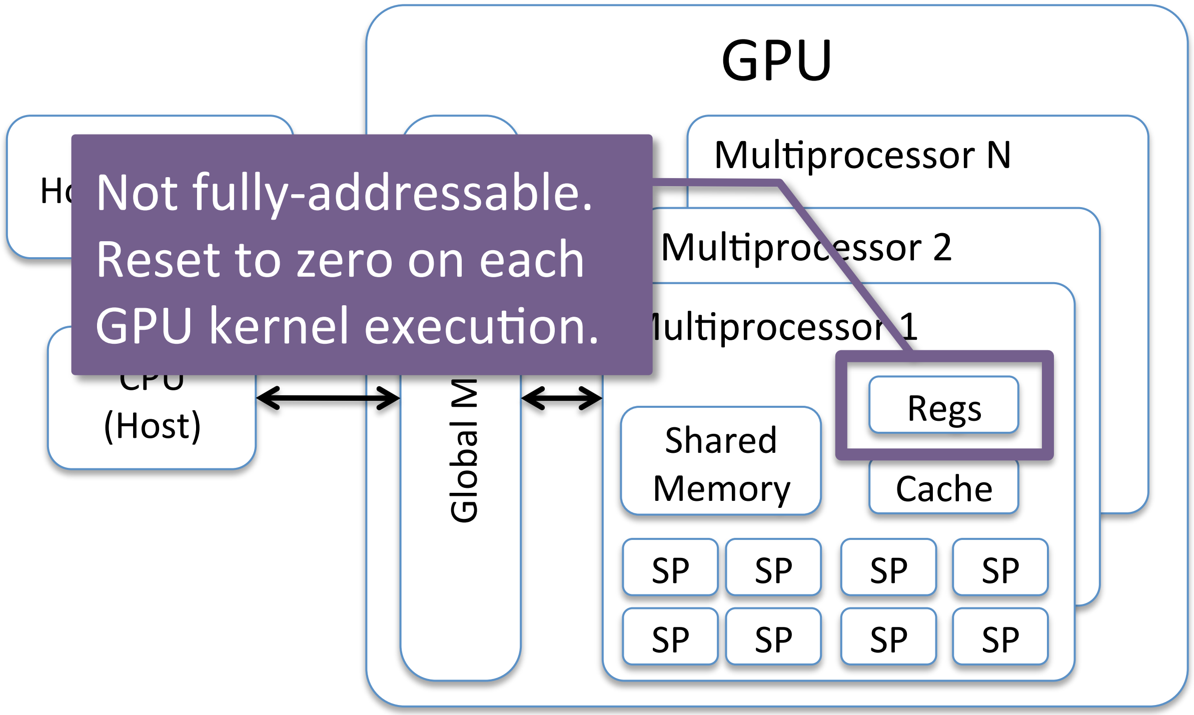
Who holds the keys?



Many different caches (L1-L3, texture, constant).
Unfortunately, the data stored there cannot be managed by the programmer

- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

Who holds the keys?



Not fully-addressable. Reset to zero on each GPU kernel execution.

- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

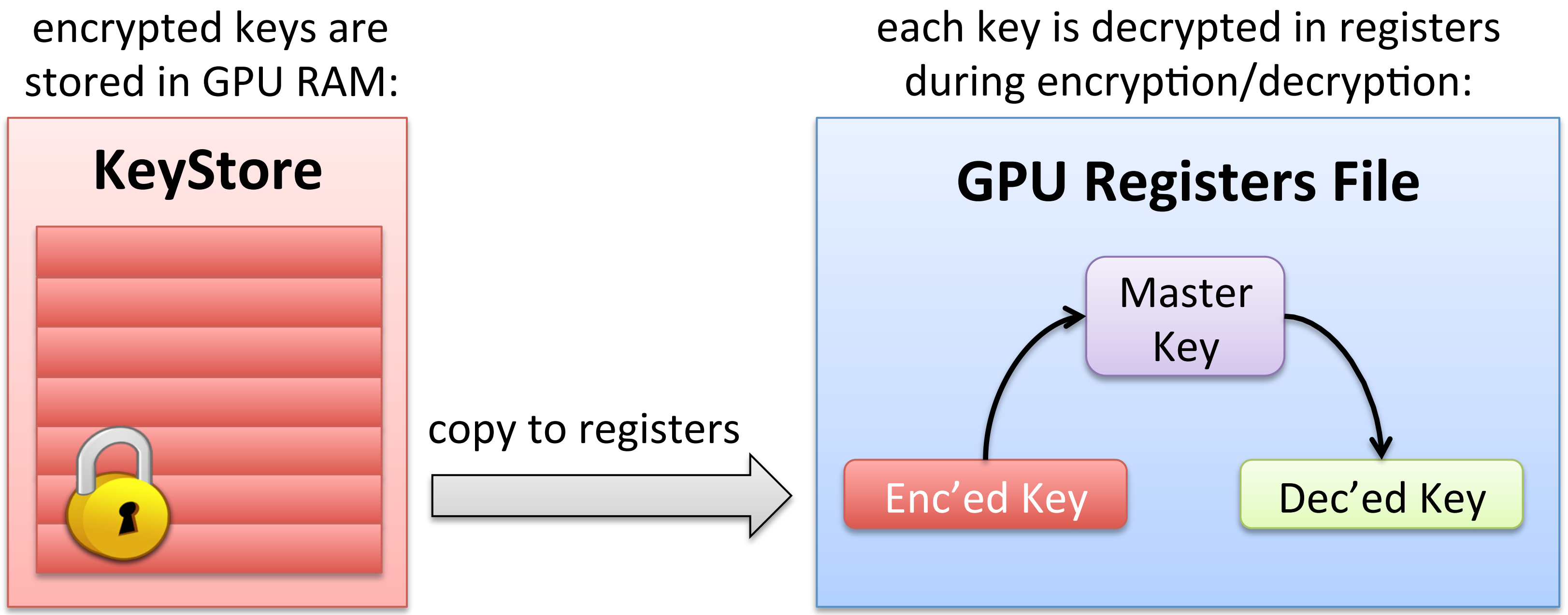
Keeping secrets on GPU registers

- Secret keys are loaded on GPU registers at an early stage of the bootstrapping phase
  - Preferably from an external storage device

- Unfortunately, the number of available registers in current GPU models is small
  - Enough for a single/few secret keys, but *what about multi-homing servers?*

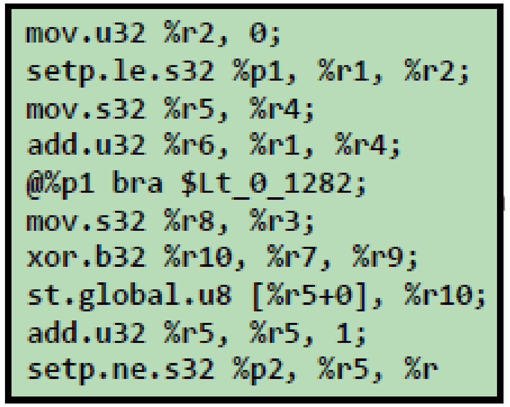
Support for an arbitrary number of keys

- We can use a separate KeyStore array that holds an arbitrary number of secret keys


encrypted keys are stored in GPU RAM:
KeyStore
copy to registers
each key is decrypted in registers during encryption/decryption:
GPU Registers File
Master Key
Enc'ed Key
Dec'ed Key

Implementation Challenges

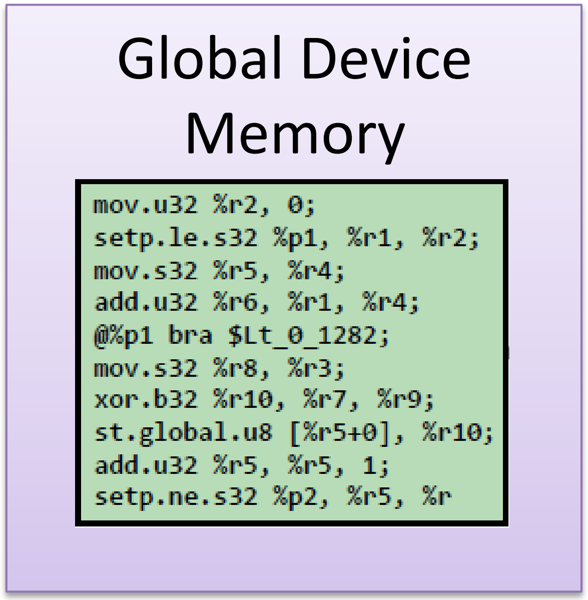
- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?



```
mov.u32 %r2, 0;
setp.le.s32 %p1, %r1, %r2;
mov.s32 %r5, %r4;
add.u32 %r6, %r1, %r4;
@%p1 bra $Lt_0_1282;
mov.s32 %r8, %r3;
xor.b32 %r10, %r7, %r9;
st.global.u8 [%r5+0], %r10;
add.u32 %r5, %r5, 1;
setp.ne.s32 %p2, %r5, %r
```

Where is the code?

- GPU code is initially stored in global device memory for the GPU to execute it
  - An adversary could replace it with a malicious version



```
mov.u32 %r2, 0;
setp.le.s32 %p1, %r1, %r2;
mov.s32 %r5, %r4;
add.u32 %r6, %r1, %r4;
@%p1 bra $Lt_0_1282;
mov.s32 %r8, %r3;
xor.b32 %r10, %r7, %r9;
st.global.u8 [%r5+0], %r10;
add.u32 %r5, %r5, 1;
setp.ne.s32 %p2, %r5, %r
```

Preventing code modification attacks

- Three levels of instruction caching (icache)
  - 4KB, 8KB, and 32KB, respectively
  - Hardware-managed
- **Opportunity:** Load the code to the icache, and then erase it from global device memory
  - The code runs indefinitely from the icache
  - Not possible to be flushed or modified

PixelVault Crypto Suite

- AES-128
- RSA-1024

AES Implementation

- The key and all intermediate states are stored in GPU registers
  - 16 bytes for the key
  - 16 bytes for the round key
  - 16 bytes for the input/output block

- The only data that is written back to global, off-chip device memory is the output block

RSA Implementation

- During exponentiation, each thread needs three temporary values of (n + 2) words each, where n is the size of the key in bits
  – 408 words for 1024-bit keys

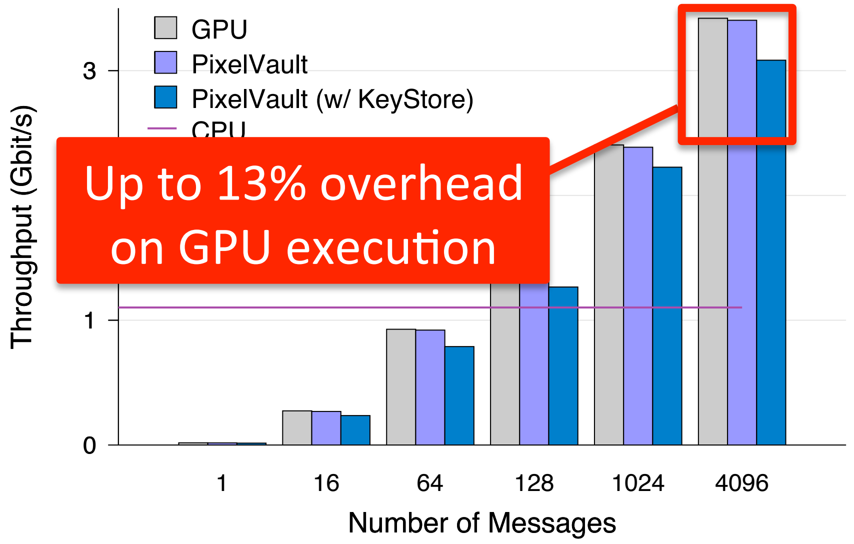
- Unfortunately, there is not always enough space to hold all three temporary values in registers
  – Store the three temporary values in shared memory (i.e. scratchpad memory)

Performance Evaluation

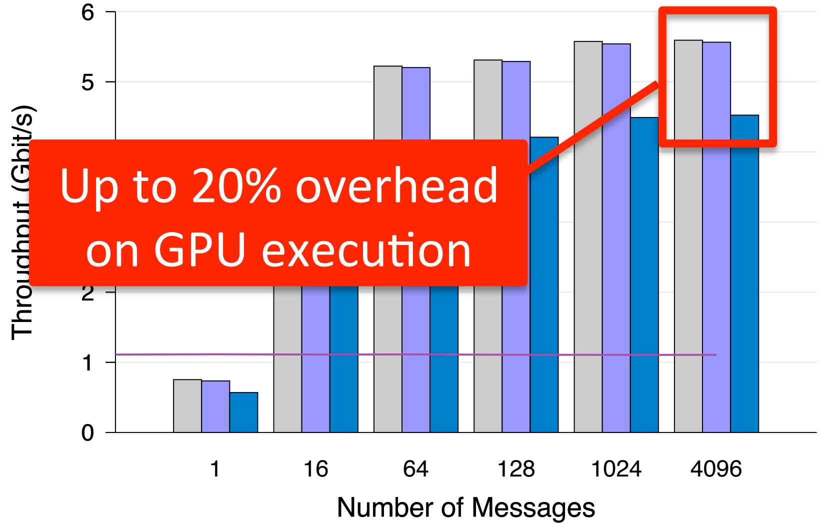
- Hardware setup
  - 2x Intel Xeon E5520 Quad-core CPUs at 2.27GHz
  - 12GB of RAM
  - GeForce GTX480

- Comparison against the standard OpenSSL implementation
  - No AES-NI support

AES-128 CBC Performance

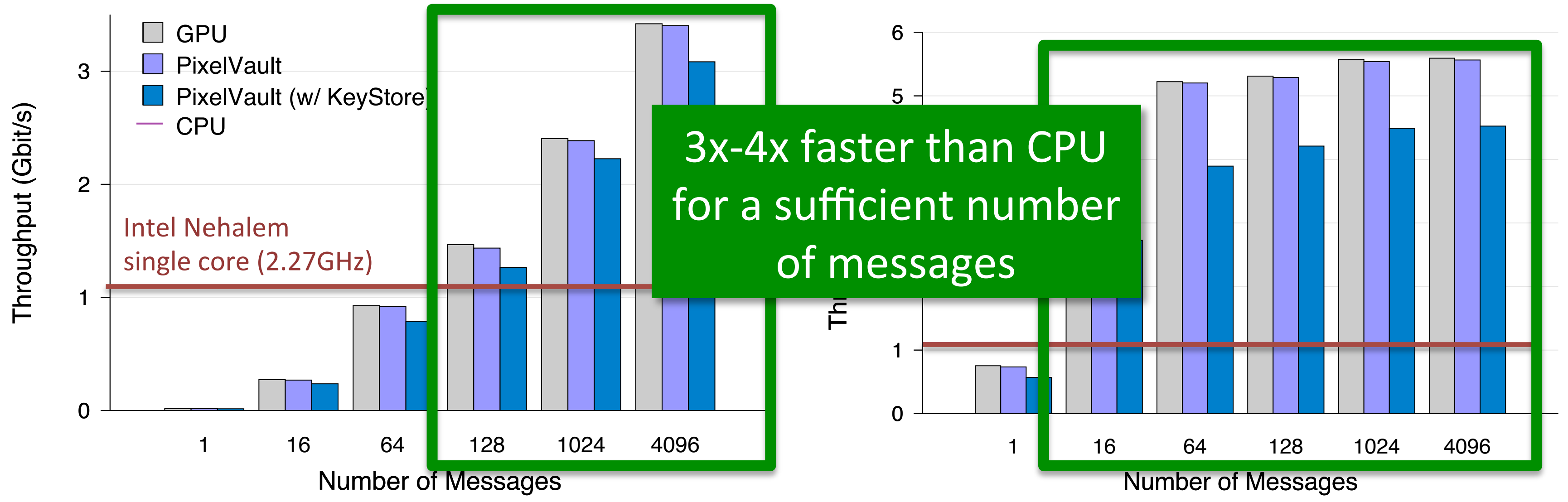


Encryption



Decryption

AES-128 CBC Performance



Encryption Decryption

RSA 1024-bit Performance

#Msgs	CPU	GPU [25]	PixelVault	PixelVault (w/ KeyStore)
1	1632.7	15.5	15.3	14.3
16	1632.7	242.2	240.4	239.2
64	1632.7	954.9	949.9	939.6
112	1632.7	1659.5	1652.4	1630.3
128	1632.7	1892.3	1888.3	1861.7
1024	1632.7	10643.2	10640.8	9793.1
4096	1632.7	17623.5	17618.3	14998.8
8192	1632.7	24904.2	24896.1	21654.4

- PixelVault adds an 1%-15% overhead over the default GPU-accelerated RSA

RSA 1024-bit Performance

#Msgs	CPU	GPU [25]	PixelVault	PixelVault (w/ KeyStore)
1	1632.7	15.5	15.3	14.3
16	1632.7	242.2	240.4	239.2
64	1632.7	954.9	949.9	939.6
112	1632.7	1659.5	1652.4	1630.3
128	1632.7	1892.3	1888.3	1861.7
1024	1632.7	10643.2	10640.8	9793.1
4096	1632.7	17623.5	17618.3	14998.8
8192	1632.7	24904.2	24896.1	21654.4

- Still faster than CPU when batch processing >128 messages

Conclusions

- Cryptography on the GPU is not only fast …
- … *but* also **secure!**
  – Preserves the secrecy of keys even when the base system is fully compromised

- Future work
  – Adapt to other ciphers and application domains
  – Apply to mobile and embedded devices

PixelVault: Using GPUs for Securing Cryptographic Operations

thank you!

Giorgos Vasiliadis	**gvasil@ics.forth.gr**
Elias Athanasopoulos	elathan@ics.forth.gr
Michalis Polychronakis	mikepo@cs.columbia.edu
Sotiris Ioannidis	sotiris@ics.forth.gr