Relatório Técnico: GPU Convex IoU

Aceleração com GPU para Cálculo de IoU em Elipses usando Aproximação por Polígonos Convexos

Autor: Gabriel
Data: Janeiro de 2026

---

1. Introdução

1.1 Problema

O cálculo de IoU entre elipses utilizando o shapely é extremamente lento, já que a etapa de avaliação pode conter milhões de interseções a serem calculadas. Como o shapely utiliza o CPU, os cálculos são sequenciais e não possuem o mesmo poder de paralelismo de um algoritmo rodado em GPU.

1.2 Solução Proposta

Este projeto implementa uma biblioteca CUDA para cálculo acelerado de IoU entre elipses, usando aproximação por polígonos convexos. A abordagem mais robusta oferece:

- Speedup de **240x** no tempo de cálculo de IoU puro (quando comparado ao shapely)
- **Precisão equivalente** (diferença máxima de 0,02%)

---

2. Fundamentação Teórica

2.2 Aproximação por Polígono

A primeira etapa foi criar um código capaz de gerar polígonos a partir de elipses,

no qual cada elipse é aproximada por um polígono convexo com N pontos uniformemente distribuídos.

2.3 Cálculo de IoU dos Polígonos

Após ter a capacidade de gerar os polígonos, foram utilizados arquivos ConvexIoU disponibilizados na biblioteca MMRotate.

A interseção é calculada utilizando o algoritmo de sutherland-hodgman, por ter eficiência em janelas de recorte convexas.

3. Implementação

3.1 Arquitetura

```
gpu_convexIoU/
├── device_iou.cuh      # Funções CUDA device (operações geométricas)
├── convexiou_cuda.cu   # Kernels CUDA
├── pybind_wrapper.cpp  # Bindings Python via pybind11
├── setup.py            # Configuração de build
└── README.md           # Documentação
```

3.2 Métodos de IoU explorados e implementados:

3.2.1 Kernel por par

A primeira tentativa de implementação utilizou um método direto em que a gpu calcula de forma sequencial, parecida com o cpu, o resultado foi de um tempo superior ao shapely.

3.2.2 Kernel de Matrizes NxN

O cálculo de matrizes NxN funciona de forma a comparar todas as elipses contra todas. Pode ser útil em situações em que o detector gera muitas caixas e precisa determinar qual é a mais adequada.

O cpu, por ser sequencial, seria extremamente lento nesse cenário, pois exige um número de cálculos muito grande.

3.2.3 Kernel Retangular (Por Imagem)

Possui uma matriz de IoU det x gt (detecções x ground truth). Compara todas as detecções com N ground truths. É um avanço considerável em velocidade quando comparado aos outros métodos utilizados, mas ainda tem o problema de possuir um call por imagem, adicionando tempo considerável de overhead.

3.2.4 Kernel Batched Retangular (Todas as Imagens)

Este kernel processa todas as imagens em uma única “batch”, eliminando o overhead de lançamento de kernels. Foi a solução mais veloz que consegui alcançar nas minhas tentativas

3.3 Snippets de código das Implementações

3.2.3

```
__global__ void iou_rect_kernel(
      const EllipseData *d_rows,    // Detecções
      const EllipseData *d_cols,    // Ground Truths
      float *matrix,                // Matriz de IoU (N_det × N_gt)
      int num_rows, int num_cols, int num_points
)
```

3.2.4

```
__global__ void iou_batched_rect_kernel(
      const EllipseData *d_all_dets,   // Todas detecções concatenadas
      const EllipseData *d_all_gts, // Todos GTs concatenados
      float *d_all_results,         // Resultados
      const int *d_pair_info,       // [det_off, gt_off, out_off, n_det,
n_gt] por imagem
      int num_pairs,
      int total_computations,
      int num_points
)
```

---

4. Resultados Experimentais

O modelo treinado em DOTA foi o único que consegui encontrar sem ter que treinar um do zero. Como já tinha o dataset DIOR, rodei o modelo DOTA apenas para fins de comparação dos IoU gerados pelas detecções (threshold baixo).

4.1 Configuração do Experimento

Parâmetro	**Valor**
Dataset	DIOR

Parâmetro	Valor
Imagens de teste	2.048
Total de detecções	134.625
Média de detecções/imagem	65,7
Total de pares IoU	998.580 ~ 1M
Modelo	Rotated RetinaNet (pré-treinado DOTA)
GPU	NVIDIA RTX 3060 (CUDA 11.3)
Pontos do polígono	16

4.2 Resultados de Performance

Método	Tempo Total	Tempo IoU	Overhead
Shapely (CPU)	550,9 s	545,0 s	5,9 s
GPU Por-imagem	18,3 s	~12 s	~6 s
GPU Batched	5,1 s	2,3 s	2,8 s

4.3 Speedups

Comparação	Speedup (Tempo Total)	Speedup (IoU)
GPU Batched vs Shapely	**110x**	**240x**
GPU Batched vs GPU Por-imagem	**3,6x**	**5x**

O aumento de velocidade mais significativo foi de 240x quando comparado ao shapely, porém a implementação batched requer uma maior alteração dos arquivos de avaliação do GauCho, dificultando a integração. (No readme da pasta do projeto tem todas as instruções para a integração)

4.4 Precisão

A diferença máxima absoluta entre os resultados do Shapely e GPU foi:

```
ACCURACY max_abs_diff=0.00020000 over 11 keys
ACCURACY_DIFF EGBB-AP080 diff=+0.00020000
ACCURACY_DIFF EGBB-AP095 diff=+0.00006000
ACCURACY_DIFF EGBB-mAP   diff=+0.00004118
```

Conclusão: Diferença de apenas 0,02% - desprezível para aplicações práticas.

4.5 Tradeoff: Número de Pontos do Polígono

Pontos	Velocidade	Erro Estimado
8	Mais rápido	~1%
16	Rápido	~0,02% (recomendado)
32	Moderado	~0,005%
64	Lento	Máxima precisão

Baseado nos dados de precisão e velocidade, cheguei a conclusão de que 16 pontos seriam recomendados para melhor utilização do GPU_IoU

---

5. Integração com Detectores

5.1 Arquivos Modificados

Para integrar o GPU IoU com o GauCho, são necessárias modificações em apenas **2 arquivos**:

5.1.1 eval_map.py (~30 linhas adicionadas)

```
# Adicionar no topo do arquivo:
def _try_import_convexiou_gpu():
    try:
        import convexiou_gpu
        return convexiou_gpu
    except ImportError:
        return None

convexiou_gpu = _try_import_convexiou_gpu()



# Adicionar na função tpfp_default, após o branch 'egbb':
elif opt == 'egbb_gpu':
    if convexiou_gpu is None:
        raise ImportError("convexiou_gpu não instalado")

    rows_np = np.asarray(det_bboxes[:, :5], dtype=np.float64)
    cols_np = np.asarray(gt_bboxes[:, :5], dtype=np.float64)
    ious = convexiou_gpu.calculate_iou_rectangular_numpy_from_numpy(
        rows_np, cols_np, num_points=16
    )
```

5.1.2 dior.py (~15 linhas adicionadas)

```
# Modificar:
allowed_metrics = ['mAP', 'recall', 'egbb', 'egbb_gpu']

# Adicionar novo bloco elif:
elif metric == 'egbb_gpu':
    # ... (similar ao bloco 'egbb', mas com opt='egbb_gpu')
```

5.2 Uso

```
# Instalar a biblioteca GPU IoU
pip install /caminho/para/gpu_convex_IoU

# Executar avaliação com GPU
python tools/test.py config.py checkpoint.pth --eval egbb_gpu
```

---

6. Pontos Adicionais

6.1 Por que o Batching é Importante?

Fator	Por-imagem	Batched
Lançamentos de kernel	~22.000	1
Alocações cudaMalloc	~44.000	3
Sincronizações	~22.000	1
Overhead CPU→GPU	Alto	Mínimo

O kernel batched elimina quase todo o overhead de comunicação CPU-GPU.

6.2 Limitações

1. Memória GPU**:** Para datasets muito grandes, pode ser necessário processar em chunks
2. Precisão**:** Aproximação por polígono introduz pequeno erro
3. Dependência CUDA: Requer GPU NVIDIA com CUDA 11.0+

6.3 Melhorias Futuras

1. Implementar NMS (Non-Maximum Suppression) em GPU usando a mesma infraestrutura
2. Suporte a múltiplas GPUs para datasets massivos
3. Otimização com Tensor Cores para GPUs mais recentes
4. Versão alternativa que possa ser utilizada com placas de vídeo AMD

---

7. Conclusão

Esse projeto demonstra que o cálculo de IoU para elipses pode ser extremamente acelerado usando GPU, tornando se uma escolha indispensável quando comparada a CPU.

Resultados Principais

Métrica	Valor
Speedup vs Shapely	**240x**
Tempo para 1M pares IoU	**2,3 segundos**
Diferença de precisão	**< 0,02%**

Referências

1. GauCho: Gaussian Convex Hull for Rotated Object Detection - https://github.com/jhlmarques/GauCho
2. MMRotate: Rotated Object Detection Benchmark - https://github.com/open-mmlab/mmrotate
3. Shapely: Manipulation and Analysis of Geometric Objects - https://shapely.readthedocs.io
4. CUDA Programming Guide - https://docs.nvidia.com/cuda/
