Algorithm 1 dijkstra GPU SSSP

Input: the CSR graph data $G(V, E, W)$, source vertex s;

Output: $dist(v)$, $(v \in V)$, the weight of the shortest path from s to v;

```

function $initial(s, G(V, E, W), base, vis)$   ▷ source vertex $s$, CSR graph data $G(V, E, W)$, the starting address of each batch $base$, the update ability of each vertex array $vis$;
    for each $v \in |V|$ do
        $dist(v) \leftarrow +\infty$;   ▷ initialize $dist$ to positive infinity;
    end for
    $dist(s) \leftarrow 0$;   ▷ set the source distence to 0;
    $Es \leftarrow E$ divide with $batchsize$;
    $Ws \leftarrow W$ divide with $batchsize$;
    $base(i) \leftarrow i \times batchsize$;
    $vis(i) \leftarrow$ how many parts vertex $i$ has been divided;   ▷ initialize the times of vertex $i$ updating neighbor;

    for $i$ in each batch do
        $Gs \leftarrow (Gs \cup (V, Es(i), Ws(i)))$;
    end for
end function

function $dijkstraCudaFunc(G(V, E, W)$, $dist$, $predist$, $flag$, $base$, $batchsize$, $vis)$   ▷ $G(V, E, W)$, the initially distance array $dist$, a temporary distance array $predist$, bool tag of updating events $flag$, the starting address of each batch $base$, the number of edges are copied into video memory in each batch $batchsize$, the update ability of each vertex array $vis$;
    $u0 \leftarrow threadId$;   ▷ get the thread id;
    $offset \leftarrow blockDim$;   ▷ get the number of threads in a block;

    $u \leftarrow u0$;
    while $u<|V|$ do
        if $V(u + 1) \leq base$ then   ▷ vertex $u$ is not in vedio memory this batch;
            $u \leftarrow (u + offset)$;   ▷ reuse the thread, get to the next vertex;
            continue;
        end if
        if $V(u + 1) \leq base + batchsize$ then   ▷ vertex $u$ is not in vedio memory this batch;
            $u \leftarrow (u + offset)$;   ▷ reuse the thread, get to the next vertex;
            continue;
        end if
        if $vis(u)$ then   ▷ indicate vertex $u$ still has update ability;
            $atomicSub(\&vis(u), 1)$;   ▷ the update ability of vertex $u$ minux one;
            $l \leftarrow max(base, V(u))$   ▷ get the min neighbor index of vertex $u$;
            $r \leftarrow min(base + batchsize, V(u + 1))$   ▷ get the max neighbor index of vertex $u$;
            for each neighbor between $l$ and $r$ has edge $(u, v, w)$ do   ▷ vertex $u$ has a egde to vertex $v$ weighted $w$;
                $atomicMin(\&predist(v), dist(u) + w)$;   ▷ use the atomic opt to exclusive mutually;
            end for
        end if
        $u \leftarrow (u + offset)$;
    end while
    $\_\_syncthreads()$;    ▷ synchronize all threads in the same block;

    $u \leftarrow u0$;
    **while** $u<|V|$ **do**
        **if** $predist(u)<dist(u)$ **then**
            $dist(u) \leftarrow predist(u)$;
            $flag \leftarrow 1$;    ▷ some vertex is updated;
            $vis(u) \leftarrow (V(u+1)+part-1)/part - V(u)/part$;    ▷ re-calculate the update ability of vertex $u$;
        **end if**
        $u \leftarrow (u+offset)$;
    **end while**
**end function**

$initial(s, G(V, E, W), base, vis)$;

$host\_to\_device(dist)$;    ▷ copy the $dist$ from main memory to GPU memory;

$flag \leftarrow true$    ▷ source vertex is updated;
**for** $flag$ is $True$ **do**
    **for** $i$ in each batch **do**
        $host\_to\_device(Gs(i) = (V, Es(i), Ws(i)))$;    ▷ copy the $G(V, Es(i), Ws(i))$ (part of the graph) from main memory to GPU memory;
        $dijkstraCudaFunc(G(i), dist, predist, flag, base(i), batchsize, vis)$;    ▷ call the CUDA kernal;
    **end for**
**end for**
$device\_to\_host(dist)$;    ▷ copy the $dist$ back;

**return** $result$
```